AIAA 2013-2855

A Hybrid Multi-GPU/CPU Computational Framework for Rotorcraft Flows on Unstructured Overset Grids

Dominic D.J. Chandar* , Jay Sitaraman†
and Dimitri Mavriplis‡

The advent of General Purpose Graphics Processing Units (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 such as MPI or openMP, one is able to obtain at least one order increase in speed-up for CFD applications on stationary grids. Due to a high throughput/cost ratio, GPUs are increasingly becoming popular among the CFD community. Flow fields on helicopter systems rank among one of the most challenging, and hence are computationally demanding to simulate. Further, presence of multiple bodies moving relative to each other require use of overset grid systems, which in turn, require efficient overset grid assembly methods to support the flow solution. In this context, a computational framework for flow simulation across multiple CPU cores and multiple GPUs has been developed and tested. It is shown that the overall wall-clock time for the simulation can be considerably reduced by using multiple GPU cards. The parallel GPU framework is initially tested using both explicit and implicit schemes for the flow past a sphere. Further, a two-blade hovering rotor test case adopted from literature is used to demonstrate the capability of the code towards simulating rotorcraft flows in a short span of time.

I. Introduction

Accurate representation of the flow-field around a helicopter is essential in assessing the overall performance, especially for new designs and rotor-body interactions. Computational methods require higher degrees of freedom to achieve a reasonable level of accuracy for engineering decisions to be made in the design phase. This in turn requires huge computational resources(parallel computers) which then translate into optimizing the cost to accuracy ratio. Currently the capability to simulate rotorcraft flows across parallel processors (CPU cores) do exist,[1–3] and are being widely used for predicting rotor loads. Computer hardware has been evolving rapidly during the last couple of years. With the advent of General Purpose Graphics Processing Units (GPGPUs), the Computational Fluid Dynamics (CFD) community has absorbed this technology, and several codes have been developed either from scratch[4–6] or by using an automatic CPU to GPU porting strategy.[7,8] The rotorcraft community also began to use GPUs for computational purposes.[9–11] However, most of the existing computational codes that are used to simulate rotorcraft flows (especially using overset/overlapping grids) do not have the capability to run across multiple CPU cores and multiple GPUs simultaneously. Studies on using this hybrid CPU-GPU framework for CFD applications are limited, and hence this paper is geared towards developing such a framework and understanding how effectively one can use all possible computing resources to solve a given problem.

The present framework has evolved from the authors' previous work CU++[12]which is a framework for solving partial differential equations on structured grids on GPUs. CU++ provides the user, a higher level programming framework especially for finite-difference calculations where standard discretized expressions can directly be used in place of standard GPU *kernels*(discussed in the next section) similar to C++. However, for unstructured grid based codes, only the data structures of CU++ are used in conjunction with GPU *kernels*, as stencil operations are not straightforward. Also developed using CU++ was GPUINS,[13] a GPU based three-dimensional incompressible Navier-Stokes solver on moving overset grids. The code runs

*Postdoctoral Research Associate, Department of Mechanical Engineering, University of Wyoming
†Assistant Professor, Department of Mechanical Engineering, University of Wyoming
‡Professor, Department of Mechanical Engineering, University of Wyoming

on a single GPU and was verified using standard incompressible flow test cases such as flow past a sphere and oscillating wing under low Reynolds number flow conditions. The GPU code was able to achieve a speed-up of 5 to 15 compared to a serial CPU code depending on the level of serial CPU code optimization. In this work, a hybrid CPU-GPU framework has been developed for solving the Compressible Euler equations on overset grids. The hybrid feature enables the code to run simultaneously on a GPU and several CPU cores. Although the viscous terms in the current implementation can be turned on, only inviscid test cases using the compressible solver are demonstrated at the moment to verify the capabilities of the GPU based implementation. This paper is organized as follows: In the following section, a short description of the flow algorithm on a per CPU core/GPU basis is described followed by GPU/CPU hardware implementation specifics. Parallel partitioning using METIS and GPU-CPU load balancing is described next followed by results and discussions on parallel performance.

II. Flow Solver Algorithm

The compressible Euler equations are discretized in space on an unstructured grid using a cell-centered finite volume method, and a second order Roe's scheme[15] for the convection terms. Time marching is performed using (1) an explicit low storage three stage Runge-Kutta scheme[14](also discussed in a GPU framework[6]) and (2) an implicit backward difference scheme(BDF1/BDF2) using a Newton-Krylov approach discussed below.

Consider the standard BDF1 scheme for the spatially discretized Euler equations given by:

$$\frac{\mathbf{Q}^{n+1}-\mathbf{Q}^{n}}{\Delta t_i}+\mathbf{G}(\mathbf{Q}^{n+1})=0 \tag{1}$$

where Δt_i is the time step normalized by the cell volume and $\mathbf{G}$ is given by:

$$\mathbf{G}(\mathbf{Q}^{n+1})=\sum_{face}\mathbf{F}^{n+1}\cdot\mathbf{n}dS \tag{2}$$

where $\mathbf{F}$ is the interface flux computed using Roe's approximate Riemann solver.[15] Eq. 2 is non-linear, and several iterations are performed to march Eq. 1 from $t=t^n$ to $t=t^{n+1}$. We replace the index $n+1$ by k and the linearization of F is performed about the state k. This linearization is given by:

$$\mathbf{F}^{k+1} = \mathbf{F}^{k}+d\mathbf{F}^{k}(\mathbf{Q}_L,\mathbf{Q}_R) \tag{3}$$

$$\mathbf{F}^{k+1} = \mathbf{F}^{k}+\frac{\partial\mathbf{F}}{\partial\mathbf{Q}_L}\Delta\mathbf{Q}_L+\frac{\partial\mathbf{F}}{\partial\mathbf{Q}_R}\Delta\mathbf{Q}_R \tag{4}$$

Hence Eq. 2 becomes

$$\mathbf{G}(\mathbf{Q}^{k+1})=\sum_{face}\left(\mathbf{F}^{k}+\frac{\partial\mathbf{F}}{\partial\mathbf{Q}_L}\Delta\mathbf{Q}_L+\frac{\partial\mathbf{F}}{\partial\mathbf{Q}_R}\Delta\mathbf{Q}_R\right)\cdot\mathbf{n}dS \tag{5}$$

Denote A_L and A_R to be the exact Roe flux Jacobians with respect to the left and right states respectively, the linearized form of $\mathbf{G}$ is now given by:

$$\mathbf{G}(\mathbf{Q}^{k+1})=\mathbf{G}(\mathbf{Q}^{k})+\sum_{face}\left(A_L\Delta\mathbf{Q}_L+A_R\Delta\mathbf{Q}_R\right)\cdot\mathbf{n}dS \tag{6}$$

Instead of the *Delta* form used in standard implicit methods (where $\Delta\mathbf{Q}$ is a variable), the field variable $\mathbf{Q}$ is computed directly so that an additional *kernel*-call to compute $\mathbf{Q}$ from $d\mathbf{Q}$ can be avoided. Hence in Eq. 6 $\Delta\mathbf{Q}$ is expanded as $\mathbf{Q}^{k+1}-\mathbf{Q}^{k}$. Using the linearized flux expression Eq. 6 in Eq. 1, we obtain a *Newton* like update:

$$\mathbf{Q}^{k+1}+\Delta t_i\sum_{face}\left(A_L\mathbf{Q}_L^{k+1}+A_R\mathbf{Q}_R^{k+1}\right)\cdot\mathbf{n}dS=\mathbf{Q}^{n}-\Delta t_i\left(\mathbf{G}(\mathbf{Q}^{k})-\sum_{face}\left(A_L\mathbf{Q}_L^{k}+A_R\mathbf{Q}_R^{k}\right)\cdot\mathbf{n}dS\right) \tag{7}$$

On convergence, the index $k+1\approx n+1$. For each step k, the above system represents a linear system of the form $[B]\mathbf{Q}^{k+1}=X(\mathbf{Q}^{k},\mathbf{Q}^{n})$, and is solved using a parallel Bi-Conjugate Gradient Stabilized algorithm(BiCGSTAB)[16] outlined below. Convergence of the linear system is attained when the relative residual

Algorithm 1 Parallel BiCGSTAB on the GPU

```
Compute initial residual: r_0 = X − [B]Q_0
Communicate r_0 across various processes
Choose a vector: r'_0 = r_0
Scalars: ρ_0 = α = ω_0 = 1
Vectors: v_0 = p_0 = 0
for i = 1 to nsteps do
  !  Each process computes the next statement using NVIDIA THRUST for GPU partitions
  !  and C++ STL for CPU partitions
  ρ_i = dot(r'_0, r_{i−1})
  MPI_Reduce(rho_i)
  β = (rho_i/rho_{i−1})(α/ω_{i−1})
  !  Each process computes the following using a simple DAXPY kernel
  p_i = r_{i−1} + β(p_{i−1} − ω_{i−1} v_{i−1})
  Communicate(p_i)
  v_i = [B]p_i
  Communicate(v_i)
  α = ρ_i/dot(r'_0, v_i)
  !  Each process computes the following using a simple DAXPY kernel
  s = r_{i−1} − α v_i
  Communicate(s)
  t = [B]s
  ts = dot(t, s)
  tt = dot(t, t)
  MPI_Reduce(ts), MPI_Reduce(tt)
  ω_i = ts/tt
  !  Update solution
  Q_i = Q_{i−1} + α p_i + ω_i s
  r_i = s − ω_i t
  Communicate(r)
  if norm(r) ≤ tolerance then
    break
  end if
end for
```

falls below a tolerance. We generally converge these linear iterations to machine precision so that the outer *Newton* like iteration converges quadratically. The matrix-vector products such as $[B]Q_0$, $[B]p_i$ and $[B]s$ are evaluated as matrix-free products with the exception of the Roe flux Jacobian, where these are computed as a 5×5 matrix for each face, and multiplied with the state vector Q (or p_i, s) for each cell.

II.A. GPU Implementation

Implementing the entire algorithm on the GPU has been very straightforward as the framework and relevant data structures were already developed for the Incompressible flow code[13] and two-dimensional Euler code.[6] Programming the GPU using NVIDIA's Compute Unified Device Architecture (CUDA)[17] requires the use of *kernels* which are similar to *functions* in standard programming languages such as Fortran or C++. Each *kernel* call will spawn a number of threads depending on the type of loop involved (loop over faces or cells), and each of this thread will perform its own computation independently thereby maximizing parallel performance. The relevant *kernels* used in the present computation are as follows:

- Gradients using Green-Gauss and Fluxes: One kernel computes the face centered values (loop over faces) and another kernel sums up the face centered values and assigns it to a cell center (loop over cells).
- Runge-Kutta update: One kernel to update the field variables based on different Runge-Kutta stage constants(loop over cells).
- Newton-Krylov update: Usage of NVIDIA Thrust library[18] for computing reductions in the BiCGSTAB[16] procedure and simple DAXPY kernels to update the solution vector (loop over cells)
- Overset connectivity: kernels for donor search and classification(loop over cells).

III. Hybrid GPU-CPU Parallel Implementation

As mentioned previously, this hybrid computational framework utilizes available GPUs and CPU cores for solving a given problem. Another advantage of this hybrid framework is that the exact same code can run in parallel without GPUs without any modification to the code thereby making it more portable for non GPU based platforms. For proper load balancing, the computational grid must be partitioned in such a way that processes that run on the GPU have a higher load compared to the processes that run on CPU cores. The entire partitioning procedure is shown in Figure 1 and consists of the following steps:

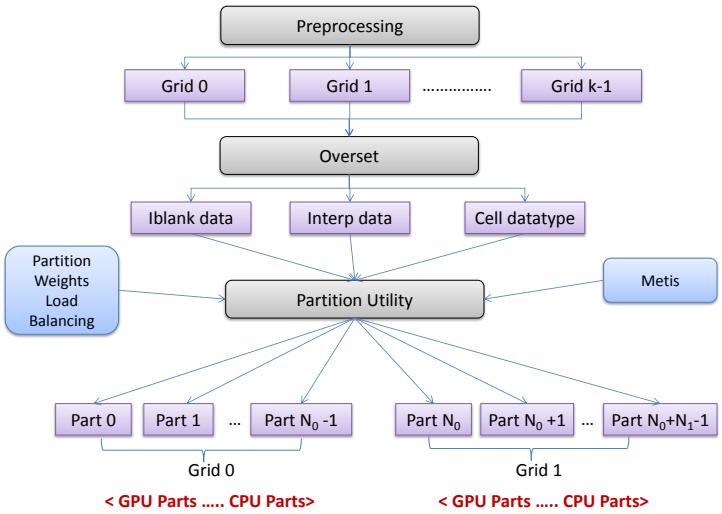

Figure 1. Grid partitioning procedure

1. For the implementation in the current paper, the overset connectivity is performed offline and the respective data written into different files. For each component grid (*not partitioned grid*), we write three pieces of information. (a) field/fringe/hole cell classification flag into the iblank file, (b) fringe and the corresponding donor cell into the interp file, and (c) overset donors and overset fringe cells into the cell datatype file.

2. As there are multiple grids, the domain partitioning must be performed for each of these grids separately. This indicates that no partition will contain more than one grid. The graph partitioning code $METIS$[19] is used to classify the cells into various partitions. As GPU partitions require more load, a *weight* file is passed as an input to $METIS$ to create weighted partitions. The partition utility calls $METIS$ for each of the grids and adds ghost cells across each partition. Figure 2 shows a portion of the partitioned grid for the hovering rotor test case. At the end of this step, the partition utility arranges the cells in a specific order for each partition as follows:

   - Cells that do not need any communication
   - Cells on a regular partition boundary that act as donors to ghost cells on neighboring partitions.
   - Overset donor cells
   - Overset fringe cells (recipient cells)
   - Ghost cells that receive data from neighboring partitions.

   Grouping the cells in this fashion enables efficient data transfer between the CPU and GPU for MPI based communication.

3. The partition utility then writes out various partition grid files which also have all the necessary information relating to communication between various processes. It also generates a process mapping file which enables mapping the partitioned grids to different processes. For each grid, the partitions are arranged such that the first few partitions are GPU partitions and the remaining are CPU partitions.

When the main flow solver runs on np processes, each process will read its respective partitioned grid based on the mapping between process ID and partition ID. Each process would know whether it is a GPU based process or a CPU based process based on the input partition file. For all GPU based processes, the relevant GPU *kernels* are called to perform various computations and for CPU based processes, the corresponding C++ functions are called. When communication is required between various processes, only the GPU based processes would transfer data between the GPU and CPU memory.

III.A. Load Balancing GPU and CPU partitions

Since a GPU runs considerably faster than a CPU core, it is important that GPU based partitions have more data to process than CPU based partitions. The ratio of the data between GPU and CPU based partitions depends on how fast a GPU performs with respect to a single CPU core. Let us assume that T is the total problem size (e.g. number of cells), s the speed-up of a GPU relative to one CPU core, n_g the number of GPUs, n_c the number of additional CPU cores (apart from those controlling a GPU), N_1 the partitioned problem size on the GPU and N_2 the partitioned problem size on the CPU. Based on these variables, the total problem size can be expressed as:

$$T = n_g N_1 + n_c N_2 \tag{8}$$

For the load to be balanced between a CPU core and GPUs, one must have:

$$N_1 = s N_2 \tag{9}$$

The above is only an assumption, as GPU speed-ups may not always vary linearly with grid size as the GPU performance depends on a wide range of parameters such as number of blocks, number of threads per block, number of registers per *kernel* and the GPU hardware itself. Using Eq.(9) in Eq.(8), we obtain

$$N_1 = \frac{sT}{n_g s + n_c} \tag{10}$$

$$N_2 = \frac{T}{n_g s + n_c} \tag{11}$$

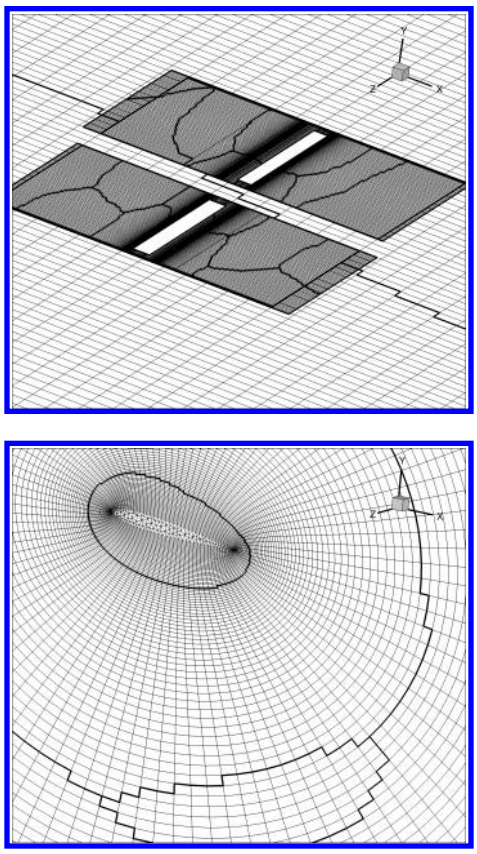

Figure 2. A section (along span and chord) of the partitioned grid for the rotor test case.

Eqs.(10) and (11) represent the load-balanced problem size on respective(GPU/CPU) partitions. To compute that however, one needs to know the quantity s, which represents the speed-up of one GPU relative to one CPU core. Hence, the problem is solved initially without any partitioning to compute the speed-up of the GPU code. More details are presented in Sec. V. One can also estimate the theoretical speed-up gained by using additional CPUs as follows:

If only GPUs are used, then the time spent by each GPU is roughly $t_{GPU1} \approx \frac{T}{n_g}$. By using additional CPU cores with load balanced, the time spent by a GPU is $t_{GPU2} \approx N_1$. Thus the speed-up gained by using additional CPU cores is given by:

$$t_{GPU1}/t_{GPU2} = 1 + \left(\frac{n_c}{n_g}\right)\frac{1}{s} \tag{12}$$

Note that the above expression does not involve any communication time. All of the computations reported in this paper were carried out on the NCAR-WY super-computing cluster. A subset of this cluster has 32 NVIDIA M2090 GPUs spanned across 16 compute nodes.

IV. Results and Discussions

IV.A. Inviscid subsonic flow past a sphere at Mach 0.3

To verify the proper implementation of the code, the flow past a sphere at nearly compressible speeds is computed, and the results are compared with inviscid incompressible flow theory with a compressibility correction factor. Figure(3) shows the contours of the pressure coefficient Cp along with a sectional plot at $z = 0$ for a single GPU computation using the explicit RK3 method in comparison with the corresponding theoretical estimates $Cp = (1 - \frac{9}{4}sin^2\theta)(1 - M_\infty^2)^{-1/2}$. Also shown in Figure (4) is a multi-GPU (24 GPUs mapped by 24 CPUs) computational result using the implicit BDF1 method. Two distinct facts can be

stated by looking at these plots. (1) The contours across the overset/partitioned boundary(indicated by the stepped edges) are continuous and (2) a good correlation between computed and theoretical estimates are obtained thereby establishing confidence in the current implementation. Convergence of the residual for the explicit and implicit methods is shown in figure 5. The implicit BDF1 formulation is about 2.4× faster than the explicit RK3 when the total wall-clock time is compared. It is hoped that, the inclusion of a preconditioner to the linear solution procedure would improve the overall convergence.

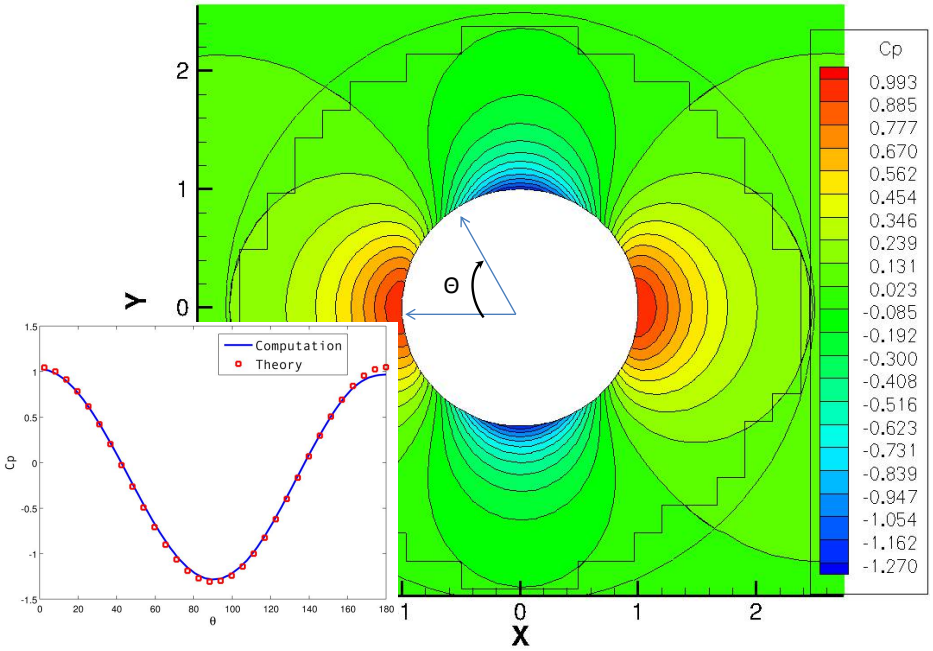

Figure 3. Contours of the pressure coefficient and sectional pressure coefficient variation (using explicit RK3 time stepping and a single GPU) in comparison with theoretical estimates for the inviscid flow past a sphere at Mach 0.3

IV.B. Inviscid transonic flow past a rotor in hover

This test case is based on the experimental work of Caradonna and Tung[20] and aids in the validation of the code for rotorcraft applications involving moving grids. The rotor consists of two untwisted, untapered wings of NACA0012 profile with an aspect ratio of 6 as shown in Figure (6). For initial computation purposes, there is no hub hence its interaction with the rotors is not accounted for. Several test cases corresponding to various rotational speeds(or tip Mach numbers) and collective pitch settings have been computed. The only difference in the implementation between the sphere test case and the rotor test case lies in the flux expressions where grid speed terms and source terms are added to account for the rotational reference frame. The governing equations are solved in the rotating reference frame but expressed as quantities in the inertial frame; cf.[21] All computations in this section were computed using an explicit RK3 time stepping scheme as the grid speed terms have not yet been accounted for in the Roe flux Jacobian. For a collective pitch of 5^0 we prescribe an angular velocity of $\omega_y = -0.1359$ about the $y - axis$ corresponding to a tip Mach number of 0.815. The entire grid consists of $1.1M$ cells with a combination of hexahedra and prisms. Using a single GPU, the computed and measured pressures coefficients ($Cp = \frac{p-p_\infty}{0.5\rho_\infty u_t^2}$)($u_t$=sectional linear velocity) at two different sections, $Z = 3$ and $Z = 5.34$ are compared in fig.(7). A reasonable comparison can be obtained with some visible differences due to the absence of viscous effects in the computation. Also plotted in the same figure are (1) surface pressure contours indicating the formation of a shock towards the tip of the rotor and (2) $Y-$velocity contours showing the downwash from the wing tips. All the preceding comparisons with experimental data are encouraging and leaves us with a lot of room for further development.

For parallel computations, the total number of cells was increased to $3M$ so that the partitioned grids

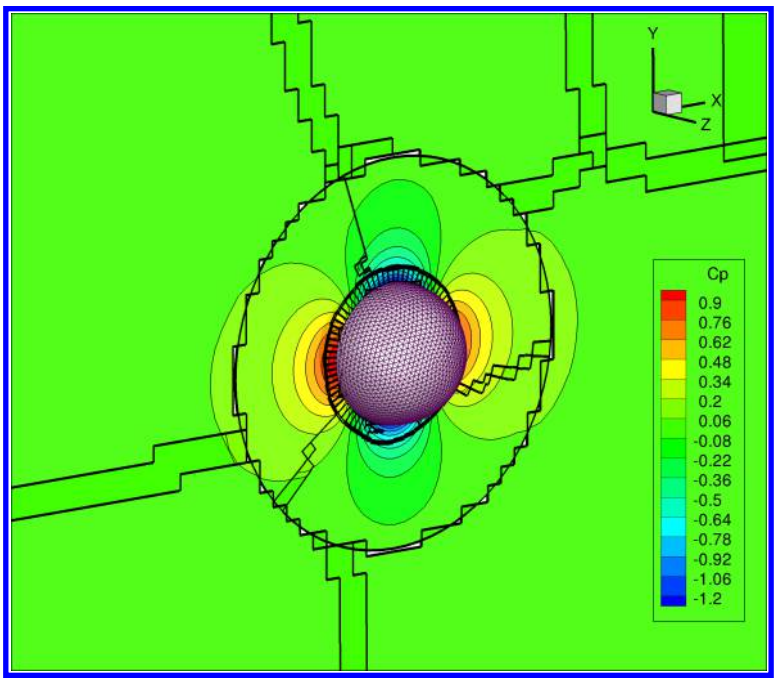

Figure 4. Contours of the pressure coefficient for the inviscid flow past a sphere at Mach 0.3 using 24 GPUs and implicit BDF1 time stepping

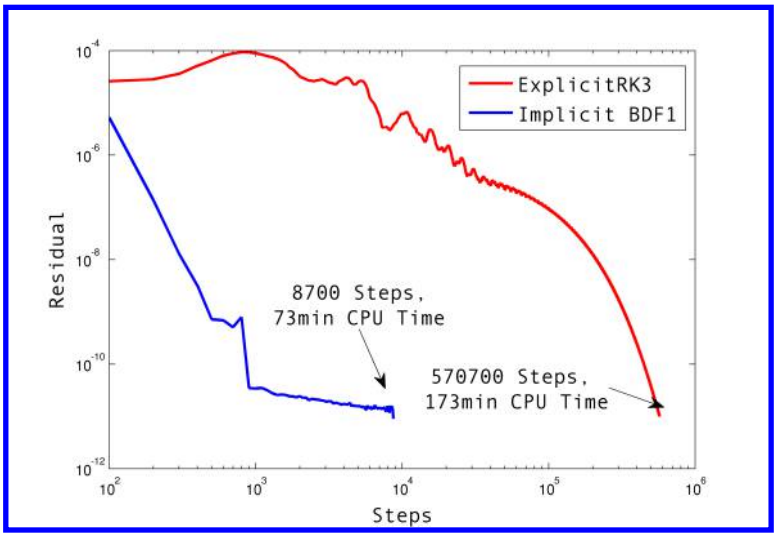

Figure 5. Convergence of the residual for the explicit-RK3 in comparison with implicit BDF-1 time stepping

that run on the GPU have sufficiently larger load. Figures (8) and (9) show the pressure contours on a section along the wing tip and the inviscid rotor wake respectively. The partition boundaries are also shown to demonstrate that the contours are continuous across these boundaries and that the communication between various partitions has taken place without any issues. As the background grid is relatively coarse in comparison with the rotor grid (Figure 2), the contours across the overset boundary have some discontinuities.

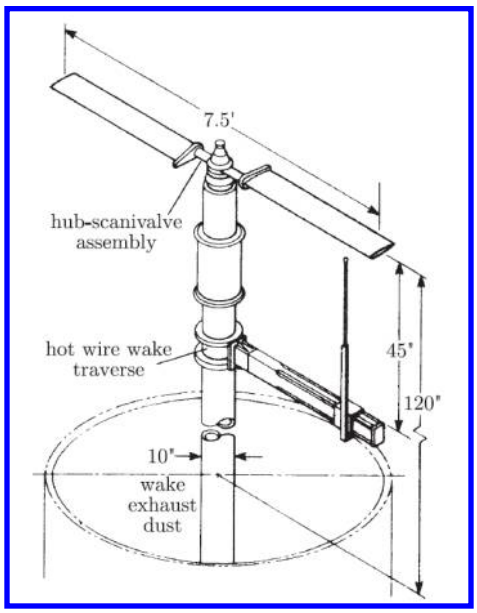

Figure 6. Experimental setup for the hovering rotor test case, adapted from Caradonna and Tung[20]

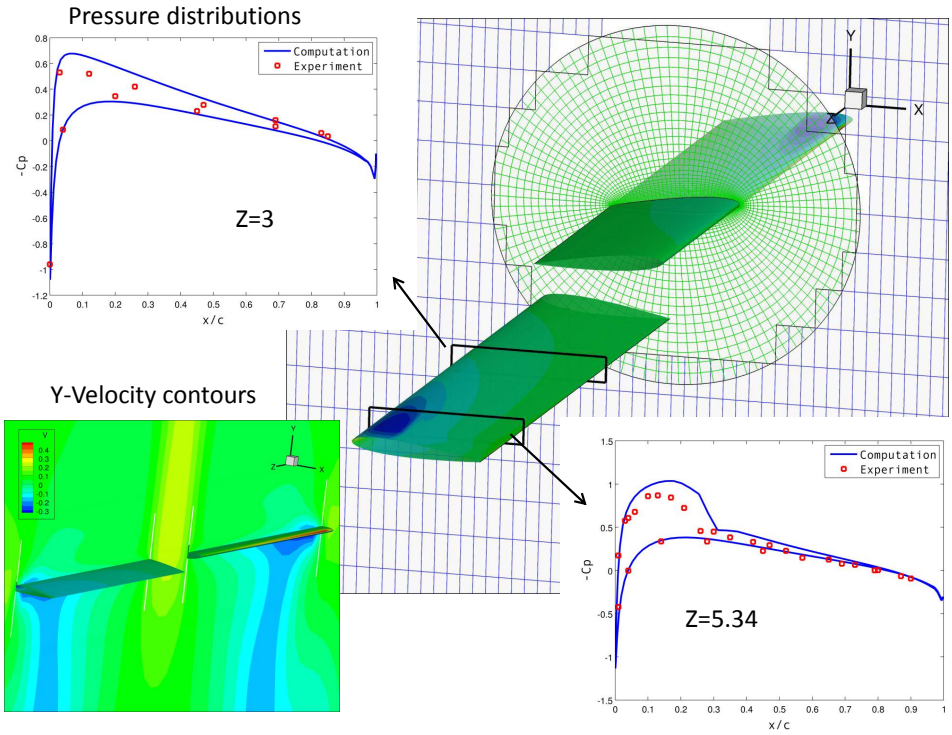

Figure 7. Computed pressure distributions and Y-velocity contours for the hovering rotor test case on a single GPU

V. Parallel GPU Performance

To assess the performance of the parallel code, computations are performed across a range of CPU cores(8,12,16, and 20) and GPUs(1,4,8,12,16, and 20). For example, if 12 GPUs are used in conjunction with

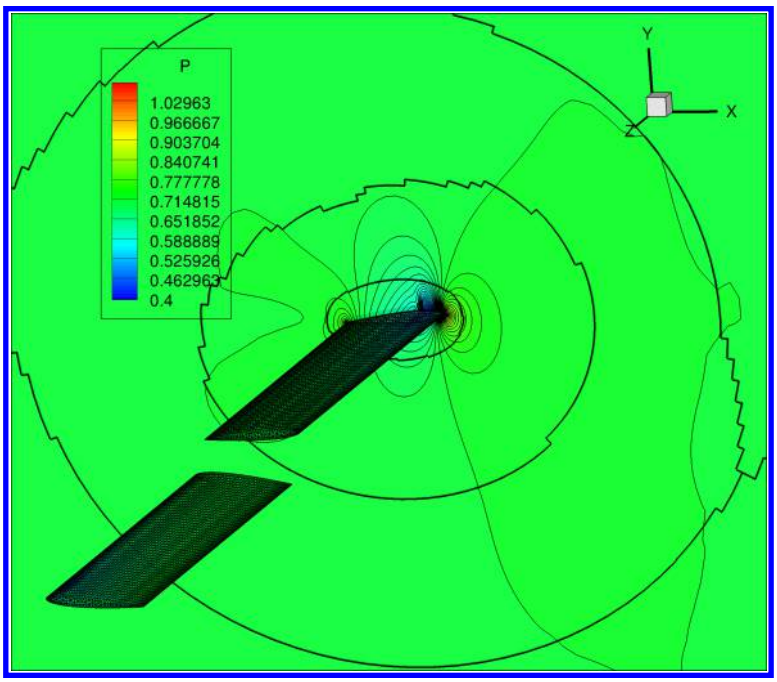

Figure 8. Computed pressure contours along $Z = -5.8$ for the hovering rotor test case on 24 GPUs and 96 CPU cores

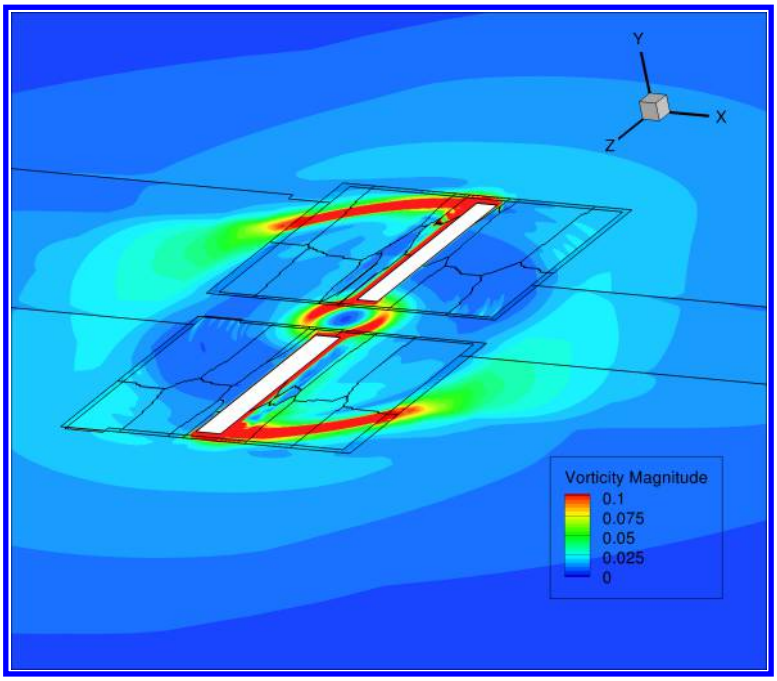

Figure 9. Computed vorticity(magnitude) contours along $Y = 0$ for the hovering rotor test case on 24 GPUs and 96 CPU cores

16 CPU cores, there are 12 CPU cores controlling 12 GPUs and 4 CPU cores running independent tasks. Hence n_g and n_c in Eq. 8 are 12 and 4 respectively. The compressible flow past a sphere using the explicit RK3 scheme described earlier is considered for computing the wall-clock times under different conditions. As both GPUs and CPUs are used, the load has to be balanced according to the method described in Sec. III.A. According to Eq. 9, this requires an estimate of the speed-up between a single GPU and single CPU core. Hence this quantity is computed initially for a wide range of grid sizes as shown in figure 10. Also plotted

in the same figure, is a variation of speed-up with threads per block. Based on the results from this plot, the speed-up s is taken to be 11 as an estimate for partitioning the grid. Overall performance of the hybrid setup is assessed with respect to the parallel performance of the same code on 8 CPU cores. Figure (11) shows contours of speed-up relative to the performance on 8 CPU cores. One can also compare this observed performance with corresponding theoretical estimates using the math outlined in Sec. III.A. If only 8 CPU cores are used, then the wall clock-time $\propto T/8$. If n_g GPUS and n_c CPU cores are used, the wall-clock time utilized by a CPU core $\propto T/(n_g s + n_c)$ (The GPU partition will also consume the same time as the load is balanced, but will have a different proportionality constant). The proportionality constant for both these cases is the same, as CPU cores are compared for both these cases. The ratio of these two quantities $(= (n_g s + n_c)/8)$ is essentially the theoretical speed-up of the hybrid GPU-CPU setup with respect to 8 CPU cores. Note that this speed-up does not involve any communication time. The corresponding theoretical plot is shown in Figure (12). Both these plots compare quite well and is very interesting in the sense that it gives us an idea on how many CPU cores can replace a GPU. For example, take a base case with 8 CPU cores and 0 GPUs. If we need a 2x improvement over 8 CPU cores, we would move along the x-axis and locate a point with the same contour value. From figures 11 and 12, this indicates that we need roughly 17 CPU cores. However one may also traverse vertically along the Y-axis and attain the same contour value with 2 GPUs and 8 CPU cores. The same can be said for other combinations of CPU cores and GPUs.

V.A. Best CPU Performance Vs. Best GPU Performance

For a given grid size, it would be beneficial to know how much potential one could extract from an array of GPUs in comparison with a standard parallel implementation that uses only CPU cores for a realistic test case such as the hovering rotor described earlier. To estimate this, we first run the code without any GPUs and estimate the wall-clock time for different number of processes. Following this run, we use as many GPUs as the number of processes so that each GPU is mapped by one CPU core. Although the maximum number of GPUs present in the NCAR-WY Supercomputing Caldera cluster is 32, only 24 could be used at a time due to job scheduling restrictions. Figure (13) shows a summary of the wall-clock times for ten iterations in *seconds* for various runs. We can see that the best CPU performance is attained using 126 CPU cores(14 cores $\times$ 9 nodes) and the best GPU performance using 24 GPUs(2 cores $\times$ 12 nodes). If cost for each run is computed as $Number\ of\ Cores \times wall-clock\ time \times (cost(\$)\ per\ core\ per\ unit\ time = k)$, we can see that the best GPU performance is 13 times economical(in terms of cost) than the best CPU performance. Here we have assumed that k is the same for both GPU and CPU based nodes as in the NCAR-WY Yellowstone cluster. The hybrid GPU-CPU implementation is not included due to the fact that when the number of GPUs used are high, a higher number of CPU cores must be used for proper load balancing (according to Eq.(12). For 24 GPUs, to get at least 50% increase in the speed-up, we need an additional 120 CPU cores. Also for a $3M$ grid the partitioned size of the grid would be small for the GPU to produce any significant speed-up and that communication times may be in the same order of magnitude as that of computation times. Hence for the hybrid GPU-CPU framework to produce meaningful results, the grid has to be very large.

VI. Conclusions

We have presented an overset grid based flow computation strategy on multi-GPU, multi-core architectures. Successful application and validation was demonstrated for the (1) sphere test case and (b) Caradonna and Tung rotor. It is also demonstrated how the hybrid nature of the code where both GPUs and CPU cores are active, provide additional speed-up than cases that involve only GPUs. In future, we plan to:

- Extend this framework so that the overset connectivity would be performed in parallel. This would pave way for solving problems that involve relative mesh motion such as rotor-body/rotor-rotor interaction.
- Improve the discretization procedure for increased accuracy of the simulations
- Implement viscous effects and
- Run the codes on the Kepler GPU architecture[22] to take advantage of NVIDIA $GPUDirect^{TM}$ - a framework to transfer data between various GPUs that lie across different compute nodes in a cluster without the CPU interference.

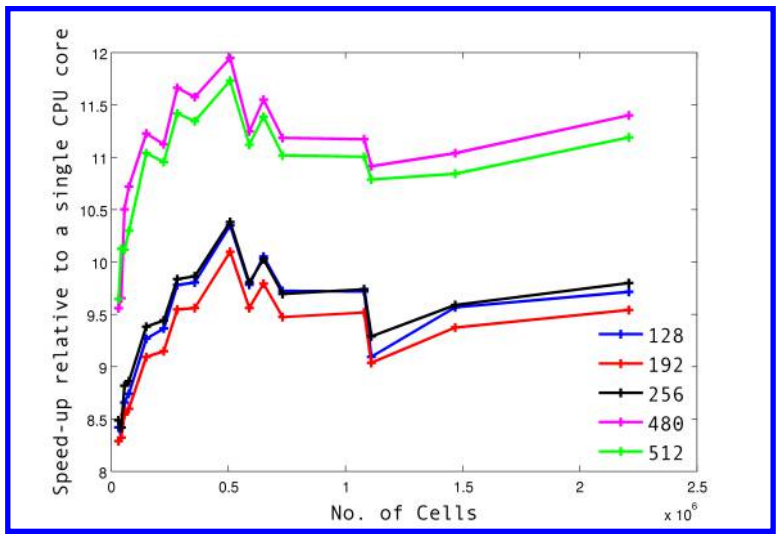

Figure 10. Speed-up(s) of one GPU relative to One CPU core

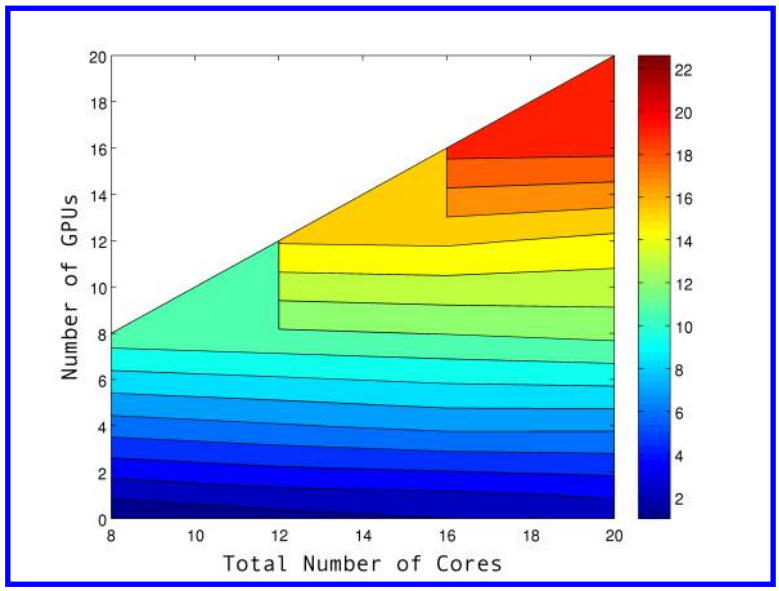

Figure 11. Actual performance of the parallel GPU code relative to 8 CPU cores

VII. Acknowledgments

We gratefully acknowledge the support from the Office of Naval Research under ONR grant N00014-09-1060 and the NCAR-Wyoming Supercomputing facility for the computational resources.

References

[1]Sankaran, V., Sitaraman, J., Wissink, A., Datta. A., Jayaraman, B., Potsdam, M., Mavriplis, D., Yang, Z., O'Brien, D., Saberi, H., Cheng, R., Hariharan, N., and Strawn, R., " Application of the Helios Computational Platform to Rotorcraft Flowfields", Paper AIAA 2010-1230, 48^{th} AIAA Aerospace Sciences Meeting and Exhibit, Orlando, FL, Jan 2010.

[2]Schwarz, T., "Simulation of the Unsteady Flow Field Around a Complete Helicopter with a Structured RANS Solver", *High Performance Computing on Vector Systems*, Springer Verlag, Heidelberg, 2006, pp. 125–137.

[3]Allen, C. B., "Parallel Flow-Solver and Mesh Motion Scheme for Forward Flight Rotor Simulation", Paper AIAA 2006-3476, 24^{th} AIAA Applied Aerodynamics Conference, San Francisco, CA, 2006.

[4]Cohen, J. M., and Molemaker, M. J., "A Fast Double Precision Code using CUDA ", *Proceedings of Parallel CFD*,

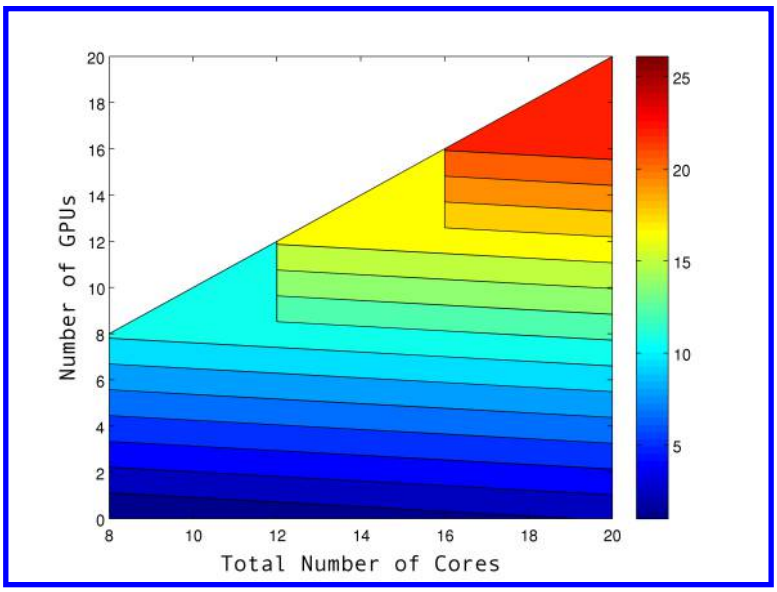

Figure 12. Theoretical performance of the parallel GPU code relative to 8 CPU cores

Number of CPU Cores/GPUs	6		12		24		48		96		126		150	
	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost	Time	Cost
CPU Only	14.1	85k	7.4	88k	4	96k	2.4	115k	1.28	122k	1	126k	1.35	202k
GPU	1.33	8k	0.73	8.7k	0.4	9.7k	-	-	-	-	-	-	-	-

Figure 13. Wall-clock times and total cost under different run conditions

Moffett Field, CA, 2009.

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

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

[7]Corrigan, A., Camelli, F., Lohner, R., and Mut, F., "Semi-Automatic Porting of a Large-Scale Fortran CFD Code to GPUs", *International Journal for Numerical Methods in Fluids*, Vol. 69 (6), 2011, pp. 314–331.

[8]Poole, D., "Introduction to OpenACC Directives", *NVIDIA GPU Technology Conference*, 2012.

[9]Lohry, M. W., Ghosh, S., and Rajagopalan, R. G., "Graphics Hardware Acceleration for Rotorcraft Brownout Simulation", Paper AIAA 2011-3224, 20^{th} AIAA Computational Fluid Dynamics Conference, Honolulu, HI, 2011.

[10]Thomas, S., Amiraux, M., and Baeder, J. D., "A GPU Accelerated Navier-Stokes Solver for Rotorcraft Applications", American Helicopter Society 69th Annual Forum, Phoenix, AZ, May 2013.

[11]Stock, M. J., Gharakhani, A., and Stone, C. P., "Modeling Rotor Wakes with a Hybrid OVERFLOW-Vortex Method on a GPU Cluster", Paper AIAA 2010-4553, 28^{th} AIAA Applied Aerodynamics Conference, Chicago, IL, 2010.

[12]Chandar, D., Sitaraman, J., and Mavriplis, D., "CU++ET: An Object Oriented Tool for Accelerating Computational Fluid Dynamics Codes Using Graphical Processing Units", Paper AIAA 2011-3222, 20^{th} AIAA Computational Fluid Dynamics Conference, Honolulu, HI, 2011.

[13]Chandar, D., Sitaraman, J., and Mavriplis, D., "Dynamic Overset Grid Computations for CFD Applications on Graphics Processing Units", Paper ICCFD7-12-2, Proceedings of the International Conference on Computational Fluid Dynamics, Big Island, Hawaii, 2012.

[14]Kennedy, C. A., Carpenter, M. H., and Lewis, R. M., "Low-Storage, Explicit Runge-Kutta Schemes for the Compressible Navier-Stokes Equations", NASA/CR 1999 209349, 1999.

[15]Roe, P.L., *Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes*, Journal of Computational Physics, Vol. 135, pp. 250-258, 1997.

[16]Van der Vorst, H. A., *Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG For the Solution of Nonsymmetric Linear Systems*. SIAM Journal on Scientific and Statistical Computing, Vol. 13, pp.631-644, 1992.

[17]NVIDIA CUDA C programming Guide 5.0, *https* : $//developer.nvidia.com/what - cuda$

[18]http://research.nvidia.com/news/thrust-cuda-library

[19]Karypis, G., and Kumar, V., "A Fast and Highly Quality Multilevel Scheme for Partitioning Irregular Graphs", *SIAM Journal on Scientific Computing*, Vol. 20 (1), 1999, pp. 252–399.

[20]Caradonna, X. C., and Tung, C., "Experimental and Analytical Studies of a Model Helicopter Rotor in Hover", NASA TM 81232, 1981.

[21]Biedron, R. T., and Thomas, J. L., "Recent Enhancements to the FUN3D Flow Solver for Moving-Mesh Applications", Paper 2009-1360, AIAA Aerospace Sciences Meeting and Exhibit, Orlando, FL, 2009.

[22]NVIDIA's Next Generation CUDA Compute Architecture, KEPLER G110, *http* : $//www.nvidia.com/content/PDF/kepler/NVIDIA - Kepler - GK110 - Architecture - Whitepaper.pdf$