Unsteady Turbulent Simulations on a Cluster of Graphics Processors

Everett H. Phillips[1], Roger L. Davis[2], and John D. Owens[3]
University of California, Davis, CA, 95616

This paper describes the GPU accelerated MBFLO2 multi-block turbulent flow solver completely in double precision using CUDA and the latest generation of GPU processors. On a cluster of 8 Tesla C2050 "Fermi" GPUs and Intel Xeon X5550 "Nehalem" quad-core CPUs, we achieve 9x speedup over the parallel CPU solver or 70x speedup over the serial solver. High performance is obtained by optimizing the data layout on the GPU, optimizing data transfers and using asynchronous memory copies to overlap GPU execution with communications. We test the solver on a turbulent flat plate and an unsteady turbulent cylinder with 3.2 million grid points. We confirm the GPU results are in agreement with turbulent flow theory. We discuss the GPU optimization techniques used to reach this level of performance.

Nomenclature

E	=	total energy
g	=	acceleration due to gravity
H	=	total enthalpy
h	=	static enthalpy
I	=	rothalpy
k	=	turbulent kinetic energy
p	=	pressure
Pr	=	Prandtl number
Pr_t	=	turbulent Prandtl number
R	=	radius from specified axis of rotation
S_{ij}	=	mean strain-rate tensor
u	=	axial velocity component
$\hat{u}$	=	internal energy
v	=	tangential velocity component
V	=	velocity magnitude
z	=	elevation
μ	=	coefficient of viscosity
μ_τ	=	turbulent coefficient of viscosity
ω	=	turbulent dissipation rate divided by turbulent kinetic energy
Ω	=	rotational velocity about specified axis of rotation (rads/s)
τ_{ij}	=	shear stress tensor
ρ	=	density

[1] Graduate Researcher, Mechanical and Aerospace Engineering, AIAA Student Member, ehphillips@ucdavis.edu
[2] Professor, Mechanical and Aerospace Engineering, AIAA Associate Fellow, davisrl@ucdavis.edu
[3] Associate Professor, Electrical and Computer Engineering, owens@ece.ucdavis.edu

I. Introduction

HIGH-PERFORMANCE automated design optimization procedures require increasing accuracy of aerodynamic performance predictions via state-of-the-art turbulence modeling techniques and quick turn-around times in order to be viable for next generation designs and differentiate between configurations[1]. The computing resource required for such tools are prohibitively expensive. We investigate the use of Graphics Processing Units (GPUs) to accelerate turbulent flow simulations and achieve order-of-magnitude speedups which have the potential to change the way CFD is used in the design process.

In the current work we build on our previous work[2] in which we demonstrated GPU cluster acceleration of the viscous flow capability in the MBFLO2 solver using single precision computations on both G92 and GT200 class GPUs, with speedups of up to 8x over the parallel CPU solver on Core2Quad architecture CPUs . In the current effort we extend this work to include turbulent flows using double precision on GF100 "Fermi" class GPUs. After applying several optimization techniques to maximize the performance of the code, we reach 9x speedup over the latest generation "Nehalem" based CPUs.

It's important to note that since our previous work[2] uses older model GPUs and CPUs and single precision computations, our current results cannot be directly compared to those previously obtained. The fact that we can reach similar levels of performance is due to additional GPU optimizations and our use of the latest "Fermi" Tesla C2050 GPUs, which are designed for double precision performance

A. Previous Work

The use of programmable graphics processors to accelerate non-graphics tasks[3] has grown tremendously over the last five years. Many of these researchers implement fluid simulations, however, most are limited to the 2D incompressible Navier-Stokes equations using pressure projection methods[5-9], which are interesting from a performance and simulation point of view, but use simplified numerics which lack the proper accuracy required for engineering design applications. Furthermore, negelecting the effects of compressibility leads to much simpler formulations which are not applicable to transonic and supersonic flows of interest. Our work, in contrast, utilizes complex state-of-the-art compressible flow numerical techniques designed for efficient and accurate aerodynamic performance prediction.

Later works by Hagen et al.[10] began to incorporate more advanced numerical methods such as high-resolution finite volume with Runga-Kutta time integrations, and study compressible flows with both 2D and 3D Euler equations accelerated by the G70 GPU. They use Cartesian meshes which unfortunately limits the application to simple geometries. Increases in geometric complexity came about in the work of Brandvik and Pullan[11] who solve both 2D and 3D Euler equations on single block structured grids. Using a single G80 or R500 series GPU, they accelerate their computation by up to 29x over a 2.4 GHz Core2Duo CPU.

The most complex work thus far is that of Elsen et al.[12] who accelerate the 3D Euler portions of a multi-block structured grid solver NSSUS with a G80 GPU. Similar to our method, they employ a generalized multi-block grid which can conform to arbitrary complex geometry. They also use multi-grid acceleration, and achieve an impressive 15x-20x speedup on complex engineering meshes with up to 1.5 million grid points using single precision.

In contranst, our work supports distributed cluster level computing on multiple GPUs with more complex viscous and turbulent flow capabilities. We must note that although our latest solver is for 2D and axi-symmetric configurations, widely used in the design process, the same methods could be easily applied to the 3D case as well.

II. Governing Equations

The unsteady, Favre-averaged governing flow-field equations for an ideal, compressible gas in the right-handed, Cartesian coordinate system using relative-frame primary variables can be written as:

Conservation of Mass: $$\frac{\partial \rho}{\partial t} + \frac{\partial(\rho u_i)}{\partial x_i} = 0 \qquad (1)$$

Conservation of Momentum: $$\frac{\partial(\rho u_i)}{\partial t} + \frac{\partial(\rho u_j u_i)}{\partial x_j} + \frac{\partial p}{\partial x_j} = \frac{\partial \tau_{ji}}{\partial x_j} - \overline{Sm_\iota} \qquad (2)$$

Conservation of Energy: $$\frac{\partial E}{\partial t} + \frac{\partial(\rho u_j I)}{\partial x_j} = \frac{\partial}{\partial x_j}\left[u_i \tau_{ij} + \left(\frac{\mu}{Pr} + \frac{\mu_t}{Pr_t}\right)\frac{\partial h}{\partial x_j}\right] \qquad (3)$$

Total Energy: $$E = \rho\hat{u} + \frac{\rho V^2}{2} - \frac{1}{2}\rho(\Omega R)^2 \qquad (4)$$

Rothalpy: $$I = H - \frac{1}{2}\rho(\Omega R)^2 = \frac{E+P}{\rho} \qquad (5)$$

Enthalpy: $$h = H - \frac{V^2}{2} = \hat{u} + \frac{P}{\rho} \qquad (6)$$

Shear Stress: $$\tau_{ij} = (\mu + \mu_t)S_{ij} \qquad (7)$$

Mean Strain Rate Tensor: $$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) - \frac{2}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij} \qquad (8)$$

The body-force vector, S_{mi}, in the momentum equation, Eq. (2), represents any body forces per unit volume such as those due to rotation (coriolis and centripetal). Additional governing equations as developed by Wilcox[13-15] for the transport of turbulent kinetic energy and turbulence dissipation rate in regions of the flow where the computational grid or global time-step size cannot resolve the turbulent eddies can be written as:

Turbulent Kinetic Energy Transport: $$\frac{\partial(\rho k)}{\partial t} + \frac{\partial(\rho u_j k)}{\partial x_j} = \left(\tau_{ij} - \frac{2}{3}\rho k\delta_{ij}\right)\frac{\partial u_i}{\partial x_j} - \beta^*\rho k\omega + \frac{\partial}{\partial x_j}\left[\left(\mu + \sigma^*\frac{\rho k}{\omega}\right)\frac{\partial k}{\partial x_j}\right] \qquad (9)$$

Turbulent Frequency Transport: $$\frac{\partial(\rho\omega)}{\partial t} + \frac{\partial(\rho u_j\omega)}{\partial x_j} = \alpha\frac{\omega}{k}\left(\tau_{ij} - \frac{2}{3}\rho k\delta_{ij}\right)\frac{\partial u_i}{\partial x_j} - \beta^*\rho\omega^2 + \sigma_d\frac{\rho}{\omega}\frac{\partial k}{\partial x_j}\frac{\partial \omega}{\partial x_j} + \frac{\partial}{\partial x_j}\left[\left(\mu + \sigma\frac{\rho k}{\omega}\right)\frac{\partial \omega}{\partial x_j}\right] \qquad (10)$$

Sub-Grid Coefficient of Turbulent Viscosity: $$\mu_t = \frac{\rho k}{\omega} \qquad (11)$$

III. GPU Computing

The latest generation of GPU processors are very attractive for scientific and high performance computing applications: they offer high floating point performance and memory bandwidth. With the growing maturity of GPU programming languages such as CUDA[4] and OpenCL, as well as the availability of programming tools such as debuggers, profilers, and standard libraries, GPUs are now first class computing platforms.

B. Fermi GF100 GPUs

In this work we use the Tesla C2050 GPUs from NVIDIA. The recently released Telsa 20-series GPUs have added many features specifically for scientific computing, such as increased double precision performance, addition of L1 and L2 caches, and ECC memory to name a few. Table 1 shows a comparison of the performance and features of the first three generations of Tesla GPUs for reference.

Table 1. Comparison of Tesla GPU Computing Cards *GPU Performance and feature comparison.*

Model	8xx	10xx	20xx
Year	2007	2008	2010
Cores (fp units)	128	240	448
Memory Size (GB)	1.5	4	3.0/6.0
Bandwidth (GB/s)	77	102	144
Single (GFLOPS)	345	622	1030
Double (GFLOPS)	-	78	515
Shared Memory (KB)	16	16	48/16
L1 Cache (KB)	-	-	48/16
L2 Cache (KB)	-	-	768
ECC	-	-	Yes

C. CUDA-Fortran

The MBFLO solver[16] is written in Fortran, however in the timeframe of our previous work[2], in which we accelerated the laminar flow routines, CUDA was only available with a C interface, so we had to mix C with Fortran in order to add GPU acceleration. This made for a less than ideal software development situation. Since then, PGI has released CUDA support in their Fortran compiler, allowing the GPU to be programmed directly in Fortran syntax. In addition to making it easier to integrate GPU acceleration into existing Fortran codes, the Fortran interface is in some ways actually easier to use. Since arrays are strongly typed with qualifiers that tell the compiler which arrays are in GPU memory, the allocations and deallocations are done in the same way as any other variables, and data transfers between GPU and CPU memory can be done with simple assignment statements (gpu_array = cpu_array) as opposed to cudamemcpy() function calls. The latest version of the PGI compiler (10.5) has also added support for the GF100 "Fermi" GPUs and also adds asynchronous data transfer functions which can be used to overlap data transfer with CPU and GPU computations. Thus, we chose to continue the development of MBFLO using the new CUDA-Fortran interface.

IV. MBFLO Solver

A. Solution Processes Overview

The conservation equations given in Eqs. (1)-(3), (9), and (10) are solved using a Lax-Wendroff control-volume, time-marching scheme as developed by Ni[17], Dannenhoffer[18], and Davis[19,20]. Numerical solution of unsteady flows is performed with a dual time-step procedure[21], allowing for use of multiple-grid and local time-stepping convergence acceleration. These techniques are second-order accurate in time and space. The flow domain is decomposed into a multi-block grid with MPI parallelization between blocks.

For turbulent flow cases, the evolution of the flow field involves the following procedures in the main time-step loop:

- Local timestep
- Laminar viscosity coefficient
- Laminar stress
- Turbulent viscosity coefficient
    - Turbulent stress
    - Turbulent flux integration
    - Turbulent block boundary communication (MPI)
    - Turbulent dual time source term
    - Turbulent numerical dissipation
- Flux integration
- Block boundary communication (MPI)
- Dual time source term
- Numerical dissipation
- Update flow variables

All of these routines have been parallelized on the GPU except for the routines that pass information between the blocks of the multi-block domain. These block boundary communication routines make use of MPI to pass and receive information at the block boundaries to accumulate the total time-rate changes at the shared nodes.

B. GPU Approach

In the GPU accelerated version, we map each process to a single GPU. The pre-processing and initialization is still handled by the CPU. However, in the beginning of the time-step procedure, the grid and flow data is transferred to the GPU memory. Then each of the routines that was previously performed on the CPU by looping over each of the grid points or grid cells, is replaced by CUDA routines that processes all grid points or cells in parallel. During the block boundary communication steps, the boundary data is transferred to the CPU, which then calls MPI to exchange data with other processes.

C. Minimizing Data transfers

The GPU and CPU have separate memory spaces requiring data to be copied into the GPU or CPU memory before the processing can take place. Data transfers of this type occur over the PCI-express bus at a relatively slow 3-6 GB/s (compared with GPU memory bandwidth of 144 GB/s). We attempt to minimize the data transfers by moving as many routines as possible onto the GPU. Since the GPU has much higher bandwidth we gather all the boundary data into a single buffer to reduce the number and size of the memory copy to the CPU. .

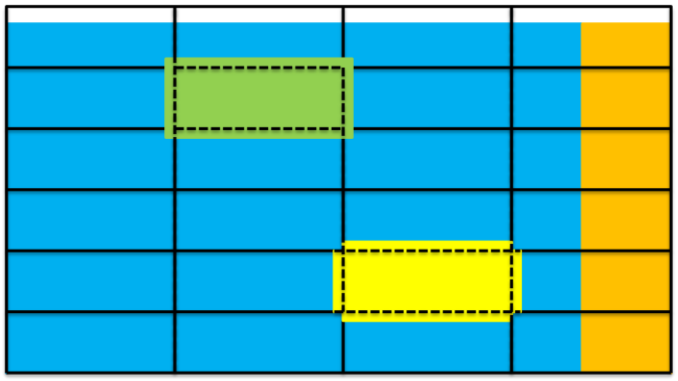

Figure 1 – GPU grid of thread-blocks to data mapping. *The array shown in blue is padded to a multiple of 32 elements shown in orange. The GPU processes data with a 2D hierarchy of thread blocks with each thread mapping to a grid point. This fine grained decomposition is done automatically when calling a GPU function called a Kernel. Two examples of data access patterns by the thread blocks are shown in green and yellow. The green shows the data needed for a thread block to compute the flux integrations, while the yellow shows data needed for a thread block to compute the smoothing or numerical dissipation.*

D. Data Layout

The original MBFLO solver stores the primary and secondary flow variables as well as turbulence variables for each grid location contiguously in memory. This facilitates the temporal locality and caching of the data when using a serial CPU processor. However, the GPU processes data in parallel with several threads accessing the same variable at a different grid location simultaneously. For best GPU memory performance, each warp or group of 32 threads should access data that is contiguous in memory. Thus, it is best to arrange the data such that each of the flow variables is stored in a separate array, with consecutive elements corresponding to the same flow variable at consecutive grid locations. In addition to contiguous data access, if the warp of threads

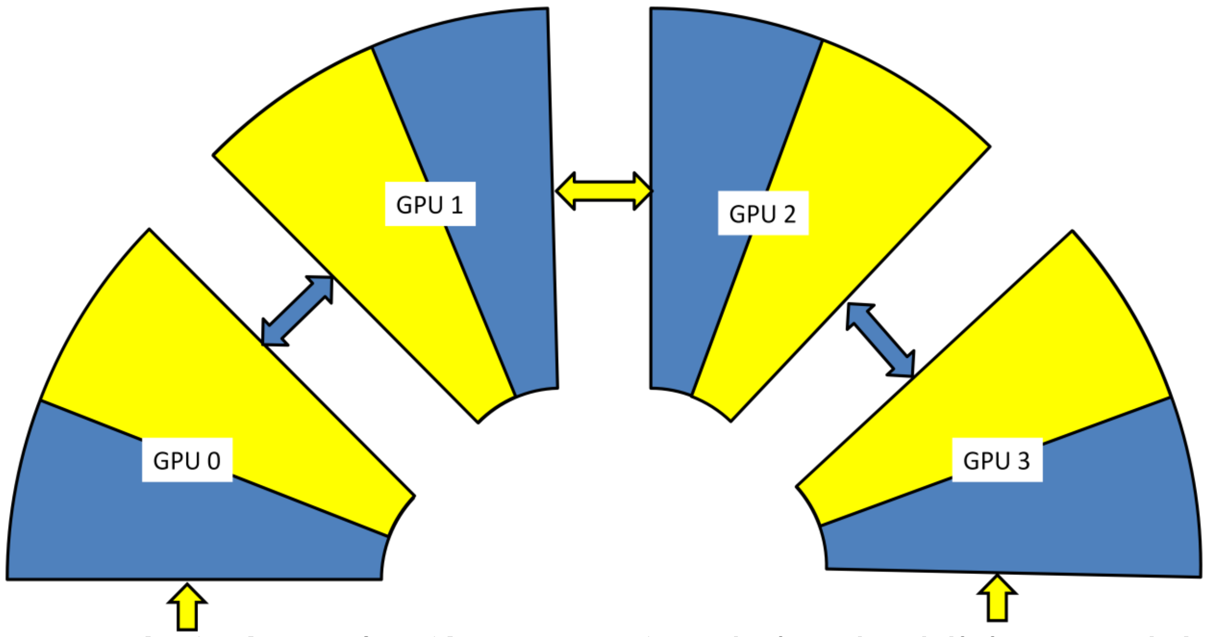

Figure 2 – Overlapping data transfers with GPU computations. *This figure shows half of an 8 GPU cylinder domain divided into blue and yellow sub-domains with arrows denoting data transfers which involve GPU-CPU transfers as well as MPI transfers over the network. The yellow domains can be computed simultaneously with communications denoted with yellow arrows, and computations in the blue regions can be overlapped with communications denoted with blue arrows. This is possible using CUDA streams with asynchronous memory transfer functions. Factoring the computations in this way resulted in up to 40% increased performance.*

accesses a segment of memory with alignment to a 128KB boundary, the data accesses are handled more efficiently and termed "coalesced". Thus, we pad the leading dimension of each array to a multiple of 32 elements, as shown in Fig. 1.

E. Shared Memory

Each of the processors in the GPU has an L1 cache that is partitioned by default into a 16KB hardware managed cache, and a 48KB programmer managed cache called shared memory. These caches are very close to the processing elements and have several orders of magnitude higher throughput and lower latency than the GPU memory. For example on the C2050, the theoretical peak memory bandwidth is 144GB/s with a latency of 400-800 clock cycles. However, the aggregate shared memory bandwidth is over 2,000 GB/s with latency of 13-16 clock cycles. Thus, in cases where threads access neighbor data as in the flux integration, stress computation, or numerical dissipation routines, we first load data into shared memory, and then processes and write results back to global memory. This reduces redundant access to global memory and can be more effective than relying on the L1 cache since we can control which data are kept in the cache.

F. Asynchronous Data Transfers

Since the block boundary communications are handled by the CPU, the data for boundaries and ghost cells must be transferred between the GPU and CPU during each time step, before and after the boundary communications routines. In our initial implementation, the GPU was idle during this time. However, by using asynchronous transfers with CUDA streams, we factor the computation of each block into multiple parts such that we can overlap the communications of one part with the computation of the other. This method is shown in Fig. 2 for a 1D domain decomposition of a cylinder flow case. This allows for greater performance and increased scalability as we can hide the communication costs. It must be noted that MBFLO does support more complex unstructured block connectivity which would require a more complex solution, such as computing only boundary points in one section, and interior points in another section.

V. Results

A. Cluster Setup

We test the solver on a cluster with 4 nodes, each equipped with 2 Tesla C2050 448-core GPUs with 3GB video memory, and 2 Intel Xeon X5550 quad-core CPUs. Thus, totally there are 8 GPUs and 8 CPUs. We use CentOS 5.3 operating system, CUDA 3.0 toolkit and R195 driver, and the PGI 10.5 Fortran Compiler.

B. Turbulent Flat Plate

The first test case is the classic flow over a flat plate at an upstream Reynolds number 1,000,000, a Mach number of 0.10, and a freestream turbulence intensity of 2%. The flow is computed on a baseline 200x400 stretched grid and run to steady state for 200,000 time steps. We use the analytical turbulent boundary layer profile to verify the computed results against theory. Figure 3 shows the agreement between the computational results and theoretical laminar-law and turbulent log-law velocity profiles.

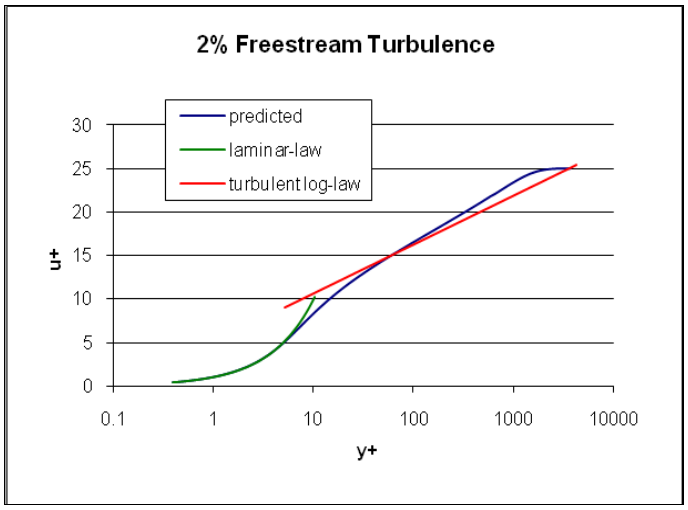

Figure 3 – MBFLO Turbulent flat plate velocity profile comparison with theory. *A validation of the MBFLO solver using a flat plate with a 200x400 grid point mesh simulating RANS turbulent flow over a flat plate at Reynolds number 1 Million and Mach number 0.1. MBFLO results are in agreement with the theoretical predictions of the laminar and turbulent profiles.*

C. Turbulent Unsteady Cylinder

The second test case is the fully turbulent flow over a cylinder at a Reynolds number 140,000, a Mach number of 0.10, and a freestream turbulence intensity of 0.6%. We use an O-Grid with 3.2 million grid points and up to 8-blocks. The instantaneous Mach contours are shown in Fig. 4.

We benchmark the performance using 1 to 8 MPI processes on the original CPU code, the initial GPU code, and the overlapping-communications GPU code that incorporates asynchronous data transfers. The test cases are run for a fixed number of iterations and then timing and solutions are compared against the serial code. Figures 5 and 6 shows the speedup of the parallel CPU and GPU results over the serial case.

On 8 GPUs the performance is increased by up to 70x over the serial CPU solver, with the naïve non-overlapping GPU code reaching 45x speedup and the parallel CPU solver reaching 7.78x speedup. Thus, if we compare the performance of parallel CPUs to GPUs the overall speedup is 9x.

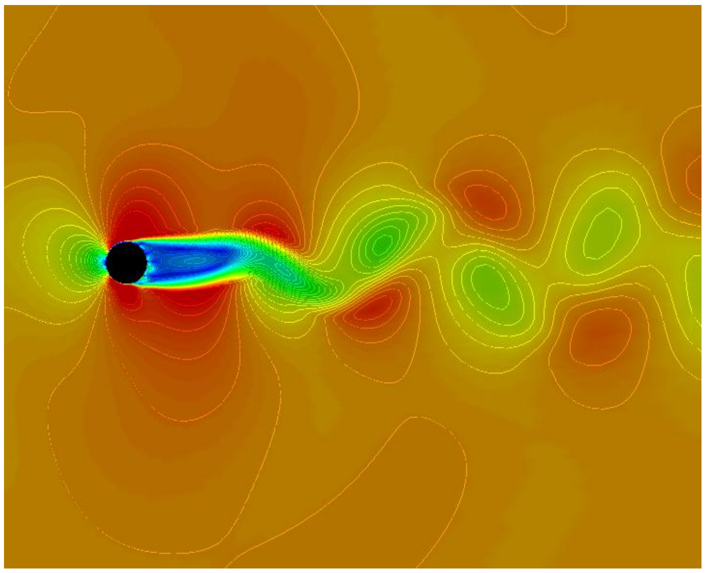

Figure 4 – MBFLO Turbulent cylinder Mach Contours. *Snapshot of the mach contours of the fully turbulent cylinder flow at Reynolds number 140,000 and Mach number 0.1*

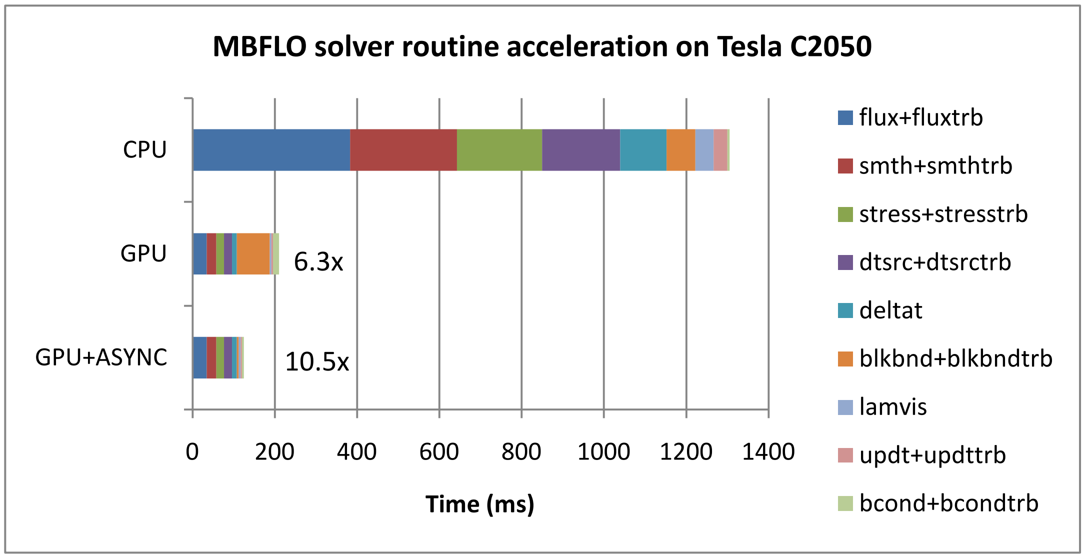

Figure 5 – Solver routine acceleration on Tesla C2050. *The solver is profiled on the turbulent cylinder flow case with a 2049x1537 grid. The CPU used is an Intel Xeon X5550 at 2.66 GHz and the GPU is the Tesla C2050 @ 1.15 GHz with ECC off. After accelerating the main computational routines the blkbnd routines, which are the only routines involving inter-block communications, become the bottleneck, accounting for approximately 40% of the execution time. After adding CUDA streams and asynchronous data transfers the blkbnd routine is overlapped with other routines and the speedup is improved from 6.3x to 10.5x***.**

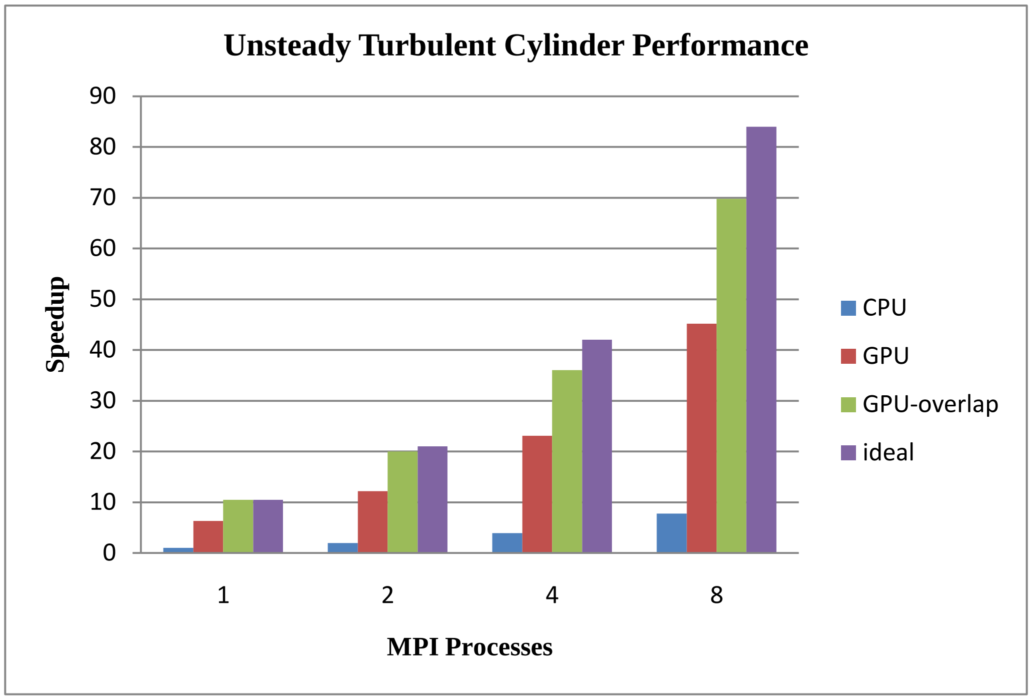

Figure 6 – MBFLO parallel performance comparison. *Performance of the solver on a 3.2M point grid for 1 to 8 nodes using CPU, GPU, and asynchronous GPU versions. Adding asynchronous data transfers allows overlapping both communications and CPU boundary treatments with GPU computations improving performance and scalability, with 8 nodes achieving 70x speedup, compared to 45x for the non-overlapped case, and 7.76x for the CPU only case.*

VI. Discussion

In our experience, the Telsa C2050 GPUs were much more flexible than the GPUs we have used in our previous work. We noticed the performance "out-of-the-box" was much better than in previous GPUs, likely due to the sharing of data through the L2 cache. However, for best results a few simple optimizations lead to even higher performance.

In order to reach optimal performance, we move as much work as possible onto the GPU to avoid costly data transfers. Even a few routines left on the CPU can cause significant slow-downs due to Amdahl's law effects. We also arrange the data layout on the GPU for optimal parallel access by warps of threads (group of 32 threads executed in SIMD). We place each variable in a separate array and pad array rows to a multiple of 32 elements. We also choose a thread-block size with a multiple of 32 in the first dimension (for example 32x4 blocks) so that warps of threads will access contiguous elements that lie in 128 KB segments of global memory, and have optimal access to shared memory which is divided into 32 banks. We make extensive use of the fast on-chip shared memory cache for stencil type computations where threads use neighbor points. Finally, we use asynchronous data transfers with CUDA streams to hide communication costs and avoid idle time.

We must also point out that the GPU requires a substantial amount of parallelism to keep all of the GPU processors occupied at all times. The GPU is designed to run tens-of-thousands of threads concurrently, thus for smaller problem sizes (less than 10,000 grid points), the cost of sending the work to the GPU and the low amount of parallelism can leave the GPU processors underutilized. However, for current and future 2D and 3D CFD applications grid densities are continually increasing and provide ample amounts of parallelism for GPU processors.

One aspect of the new Tesla C2050 processor that we did not have a chance to experiment with yet is the ability to run concurrent kernels, which brings both data-level parallelism and task-level parallelism to the GPU. This could allow for better performance on smaller problem sizes or applications that lack enough data parallelism to fill the entire GPU with a single function, but have several independent functions that can be run in parallel.

VII. Conclusion

We have demonstrated the use of the GPU cluster for compressible turbulent flow computations in double precision and show order-of-magnitude speedups in performance over traditional CPU processors. With the continued trend in performance and programmability and addition of more HPC centric features, the GPU has the power to transform the design process.

In the future, we plan to extend the GPU solver to fully 3D configurations with state-of-the-art turbulence modeling techniques and applications to automated design optimization.

Acknowledgments

The authors appreciate the sponsorship of this work by the Department of Energy's Early Career Principal Investigator Award (DE-FG02-04ER25609) and previous development work by the managers of Wright-Patterson Air Force Research Laboratory under contract 09-S590-0009-20-C1 with Dr. John Clark as technical monitor. Thanks also to NVIDIA for hardware donations.

References

[1]Davis, R. L., "Challenges for CFD in Gas-Turbine Applications," *Proceedings of CFD2006 Symposium*, Kingston, Canada, July 2006.

[2]Phillips, E. H., Zhang, Y., Davis, R. L., and Owens, J. D., "Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units," AIAA 2009-565, January 2009.

[3]Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., and Purcell, T., "A Survey of General-Purpose Computation on Graphics Hardware," Computer Graphics Forum, Vol. 26, No. 1, 2007, pp. 113.

[4]NVIDIA Corporation, "NVIDIA CUDA Compute Unifed Device Architecture Programming Guide," http://developer. nvidia.com/cuda.

[5]Bolz, J., Farmer, I., Grinspun, E., and Schroder, P., "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," ACM Transactions on Graphics, Vol. 22, No. 3, July 2003, pp. 917-924.

[6]Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G., "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware," Graphics Hardware 2003, July 2003, pp. 102-111.

[7]Harris, M. J., Baxter III, W., Scheuermann, T., and Lastra, A., "Simulation of Cloud Dynamics on Graphics Hardware," Graphics Hardware 2003, July 2003, pp. 92-101.

[8]Kruger, J. and Westermann, R., \Linear Algebra Operators for GPU Implementation of Numerical Algorithms," ACM Transactions on Graphics, Vol. 22, No. 3, July 2003, pp. 908-916.

[9]Scheidegger, C. E., Comba, J. L. D., and da Cunha, R. D., "Practical CFD Simulations on Programmable Graphics Hardware using SMAC," Computer Graphics Forum, Vol. 24, No. 4, 2005, pp. 715-728.

[10]Hagen, T. R., Lie, K.-A., and Natvig, J. R., "Solving the Euler Equations on Graphics Processing Units," Proceedings of the 6th International Conference on Computational Science, Vol. 3994 of Lecture Notes in Computer Science, Springer, May 2006, pp. 220-227.

[11]Brandvik, T. and Pullan, G., "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware," Proceedings of the 48th AIAA Aerospace Sciences Meeting and Exhibit, No. AIAA 2008-607, Jan. 2008.

[12]Elsen, E., LeGresley, P., and Darve, E., "Large calculation of the ow over a hypersonic vehicle using a GPU," J. Comput. Phys., Vol. 227, No. 24, 2008, pp. 10148-10161.

[13]Wilcox, D. C., "Formulation of the k-w Turbulence Model Revisited," AIAA Paper 2007-1408, January 2007.

[14]Wilcox, D. C., "Reassessment of the Scale-Determining Equation for Advanced Turbulence Models," *AIAA Journal*, Vol. 26, No. 11, 1988, pp. 1299, 1310.

[15]Wilcox, D. C., Turbulence Modeling for CFD, DCW Industries, Inc., La Cannada, CA, 1998.

[16]Davis, R. L. and Dannenhoffer, J. D, III, "A Detached-Eddy Simulation Procedure Targeted for Design," AIAA Journal of Propulsion and Power, Vol. 24, No. 6, Nov/Dec 2008.

[17]Ni, R. H., "A Multiple Grid Scheme for Solving the Euler Equations," *AIAA Journal*, Vol. 20, 1982, pp. 1565, 1571.

[18]Dannenhoffer, J. F., "Grid Adaptation for Complex Two-Dimensional Transonic Flows", CFDL-TR-87-10, Department of Aeronautics and Astronautics, Massachusetts Institute of Technology, August 1987.

[19]Davis, R. L., Ni, R. H., and Carter, J. E., "Cascade Viscous Flow Analysis Using The Navier-Stokes Equations," *AIAA Journal of Propulsion and Power*, Vol. 3, No. 5, September-October 1987

[20] Davis, R. L., Hobbs, D. E., and Weingold, H. D., "Prediction of Compressor Cascade Performance Using a Navier-Stokes Technique", *ASME Journal of Turbomachinery*, Vol. 110, No. 4, pp. 520-531, 1988.

[21]Jameson, A., "Time Dependent Calculations Using Multi-grid, with Applications to Unsteady Flows Past Airfoils and Wings," AIAA 91-1596, June 1991.