19th AIAA Computational Fluid Dynamics, June 22–25, San Antonio, Texas

Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware

Andrew Corrigan * Fernando Camelli † Rainald Löhner ‡

John Wallin §

George Mason University, Fairfax, VA 22030, USA

Techniques used to implement an unstructured grid solver on modern graphics hardware are described. The three-dimensional Euler equations for inviscid, compressible flow are considered. Effective memory bandwidth is improved by reducing total global memory access and overlapping redundant computation, as well as using an appropriate numbering scheme and data layout. The applicability of per-block shared memory is also considered. The performance of the solver is demonstrated on two benchmark cases: a missile and the NACA0012 wing. For a variety of mesh sizes, an average speed-up factor of roughly 9.5x is observed over the equivalent parallelized OpenMP-code running on a quad-core CPU, and roughly 33x over the equivalent code running in serial.

I. Introduction

Over the past few year GPUs (graphics processing units) have seen a tremendous increase in performance, with the latest GeForce 200 series and Tesla 10 series NVIDIA GPUs now achieving nearly one teraflop of performance, or roughly an order of magnitude higher performance than high-end CPUs [1, Sec. 1.2]. In addition to this high computational performance, the latest modern graphics hardware offers increasing memory capacity, as well as support for 64-bit floating point arithmetic. Together with CUDA,[1] which exposes GPUs as general-purpose, parallel, multi-core processors, GPUs offer tremendous potential for applications in computational fluid dynamics.

In order to fully exploit the computational power of such hardware, considerable care is required in the coding and implementation, particularly in the memory access pattern. GPUs have general-purpose global memory, which is not automatically cached and exhibits high latency in comparison with the instruction throughput of GPUs. Furthermore, with earlier CUDA-enabled GPUs, there were stringent requirements for achieving optimal effective memory bandwidth, with a large loss of performance when these requirements went unmet. With the data-dependent memory access of unstructured grid based solvers, this loss of performance is almost assured. However, with due care, *structured* grid based solvers can meet these requirements due to the regular memory access patterns of such solvers, as described in the work of Brandvik and Pullan,[2,3] and Tölke.[4] Further work on regular grid solvers includes that of Phillips et al.,[5] who have developed a 2D compressible Euler solver on a cluster of GPUs, and Thibault et al.,[6] who have implemented a 3D incompressible Navier Stokes solver for multi-GPU systems.

So far, the implementation of optimized *unstructured* grid based solvers for modern graphics hardware has been relatively rare, perhaps due to these stringent requirements. Recently Klöckner et al.[7] have implemented discontinuous Galerkin methods over unstructured grids. They achieve the highest speed-up in comparison to a CPU code in higher order cases, due to higher arithmetic intensity. An alternative means of accessing GPU memory is via texture memory, which offers automatic caching intended for memory

*Graduate Student, Computational and Data Sciences, Student Member AIAA

†Assistant Professor, Center for Computational Fluid Dynamics

‡Distinguished Professor, Center for Computational Fluid Dynamics, Member AIAA

§Associate Professor, Computational and Data Sciences

access patterns which exhibit two-dimensional spatial locality, and has been effectively used, for example, in the CUDA SDK.[8] However, this type of memory is inappropriate for the indirect memory access of three-dimensional unstructured grid solvers.

Implementing CFD solvers on graphics hardware predates CUDA. In fact, just prior to its first release, Owens et al.[9] comprehensively surveyed the field of general-purpose computation on graphics hardware (GPGPU), which included a number of primarily structured grid based solvers, such as those of Harris,[10] Scheidegger et al.,[11] and Hagen et al.[12] However, the architecture has changed substantially and many of the limitations of GPGPU via traditional graphics APIs such as OpenGL are no longer an issue.

The most recent CUDA-enabled GPUs have looser requirements for achieving high effective memory bandwidth. Roughly speaking, memory no longer needs to be accessed in a specific order by consecutive threads. Rather, high effective memory bandwidth can be achieved as long as consecutive threads access nearby locations in memory, which is called *coalescing*. Thus, if an appropriate memory access pattern is obtained, one can expect that modern GPUs will be capable of achieving high effective memory bandwidth and in general high performance for unstructured grid based CFD solvers. The purpose of this work is to study techniques which achieve this.

The remainder of the chapter is organized as follows: Section II describes the solver considered: a three-dimensional finite volume discretization of the Euler equations for inviscid, compressible flow over an unstructured grid. Section III considers the techniques used to achieve high performance with modern GPUs for unstructured grid solvers. After giving an overview of the code, techniques are described to reduce total memory access by overlapping redundant computation, increase effective memory bandwidth by using an appropriate numbering scheme, and increase effective instruction throughput by avoiding divergent branching. This is followed by a discussion of the issue of employing shared memory with unstructured grid solvers. Performance results are given in Section IV which demonstrate an order of magnitude speed-up using a GPU in comparison to an equivalent parallelized shared-memory OpenMP code running on a quad-core CPU.

II. Euler Solver

We consider the Euler equations for inviscid, compressible flow,

$$\frac{d}{dt}\int_{\Omega} \mathbf{u} d\Omega + \int_{\Gamma} \mathbf{F}\cdot\mathbf{n} d\Gamma = 0 \quad , \tag{1}$$

where

$$\mathbf{u} = \begin{Bmatrix} \rho \\ \rho v_x \\ \rho v_y \\ \rho v_z \\ \rho e \end{Bmatrix}, \underline{\mathbf{F}} = \begin{Bmatrix} \rho v_x & \rho v_y & \rho v_z \\ \rho v_x^2 + p & \rho v_x v_y & \rho v_x v_z \\ \rho v_y v_x & \rho v_y^2 + p & \rho v_y v_z \\ \rho v_z v_x & \rho v_z v_y & \rho v_z^2 + p \\ v_x(\rho e + p) & v_y(\rho e + p) & v_z(\rho e + p) \end{Bmatrix}, \tag{2}$$

and

$$p = (\gamma - 1)\rho\left[e - \frac{1}{2}\|\mathbf{v}\|^2\right]. \tag{3}$$

Here $\rho, v_x, v_y, v_z, e, p$ and γ denote, respectively, the density, x,y,z velocities, total energy, pressure and ratio of specific heats. The equations are discretized using a cell-centered, finite-volume scheme of the form:

$$\mathrm{vol}_i \frac{d\mathbf{u}_i}{dt} = \mathbf{R}_i = -\sum_{faces} \|\underline{\mathbf{s}}\| \left[\frac{1}{2}(f_i + f_j) - \beta\cdot\lambda_{\max}\cdot(u_i - u_j)\right] \tag{4}$$

where

$$f_i = \frac{\underline{\mathbf{s}}}{\|\underline{\mathbf{s}}\|}\cdot\underline{\mathbf{F}_i} \quad , \quad \lambda_{\max} = \|\underline{\mathbf{v}}\| + c, \tag{5}$$

where vol_i denotes the volume of the ith element, $\underline{\mathbf{s}}$ denotes the face normal, j is the index of the neighboring element, β is a parameter controlling the amount of artificial viscosity, and c is the speed of sound.

III. Implementation on Graphics Hardware

A. Overview

The performance-critical portion of the solver consists of a loop which repeatedly computes the time derivatives of the conserved variables, given by Eq. 4. The conserved variables are then updated using an explicit Runge-Kutta time-stepping scheme. The most expensive computation consists of accumulating flux contributions and artificial viscosity across each face when computing the time derivatives. Therefore, the performance of the CUDA kernel which implements this computation is crucial in determining whether or not high performance is achieved, and is the focus of this section.

B. Redundant Computation

The time derivative computation is parallelized on a per-element basis, with one thread per element. First, each thread reads the element's volume, along with its conserved variables from *global memory* [1, Sec. 5.1.2.1], from which derived quantities such as the pressure, velocity, the speed of sound, and the flux contribution are computed. The kernel then loops over each of the four faces of the tetrahedral element, in order to accumulate fluxes and artificial viscosity. The face's normal is read along with the index of the adjacent element, where this index is then used to access the adjacent element's conserved variables. The required derived quantities are computed and then the flux and artificial viscosity are accumulated into the element's residual.

This approach requires redundant computation of flux contributions, and other quantities derived from the conserved variables. Another possible approach is to first precompute each element's flux contribution, thus avoiding such redundant computation. However, this approach turns out to be slower for two reasons. The first of which is that reading the flux contributions requires three times the amount of global memory access than just reading the conserved variables. The second is that the redundant computation can be performed simultaneously with global memory access, as described in [1, Sec. 5.1.2.1], which hides the high latency of accessing global memory. The performance difference between each approach is stated in Section IV.

C. Numbering Scheme

In the case of an unstructured grid, the global memory access required for reading the conserved variables of neighboring elements is at risk of being highly *non-coalesced*, which results in lower effective memory bandwidth [1, Sec. 3.1]. This can be avoided however, if neighboring elements of consecutive elements are nearby in memory. In particular, if for $i = 1, 2, 3, 4$, the ith neighbor of each consecutive element is close in memory then more coalesced memory access will be achieved, as implied by the coalescing requirements for graphics hardware with compute capability 1.2 or higher [1, Page 54]. This is achieved here in two steps. The first step is to ensure that elements nearby in space are nearby in memory by using a renumbering scheme. The particular numbering scheme used in this work is the the bin numbering scheme described by Löhner [13, Sec. 15.1.2.2]. This scheme works by overlaying a grid of bins. Each point in the mesh is first assigned to a bin, and then the points are renumbered by assigning numbers while traversing the bins in a fixed order. With such a numbering in place, the connectivity of each element is then sorted locally, so that the indices of the four neighbors of each tetrahedral element are in increasing order. This ensures that, for example, the second neighbor of consecutive elements are close in memory.

The possibility of divergent branching, and thus lower instruction throughput [1, Sec. 5.1.2.2], arises since several special cases have to be considered in order to deal with faces that are on the boundaries of the computational domain. In the present case, these are marked by storing a negative index in the connectivity array that refers to the particular boundary condition desired (e.g. wing boundary, far-field, etc.). This results in possible branching, which incurs no significant penalty on modern graphics hardware, as long as all threads within a *warp* (a group of 32 consecutive threads [1, Appendix A]) take the same branch [1, Sec. 3.1, Page 14]. To minimize this penalty, in addition to having ensured that only the first face of each element can be a boundary face, the bin numbering is modified to ensure that boundary elements are stored consecutively in memory, which means that there can be at most two divergent warps.

D. Data-Dependent Memory Access and Shared Memory

Shared memory is an important feature of modern graphics hardware used to avoid redundant global memory access amongst threads within a block [1, Sec. 5.1.2]. The hardware does not automatically make use of shared memory, and it is up to the software to explicitly specify how shared memory should be used. Thus, information must be made available which specifies which global memory access can be shared by multiple threads within a block. For structured grid based solvers, this information is known a priori due to the fixed memory access pattern of such solvers. On the other hand, the memory access pattern of unstructured grid based solvers is data-dependent. With the per-element/thread based connectivity data structure considered here, this information is not provided, and therefore shared memory is not applicable in this case. However, as demonstrated by Klöckner et al.,[7] in the case of higher-order discontinuous Galerkin methods with multiple degrees of freedom per element, the computation can be further decomposed to have multiple threads process a single element, thus making shared memory applicable.

IV. Results

The performance of the GPU code was measured on a prototype NVIDIA Tesla GPU, supporting compute capability 1.3, with 24 multiprocessors. The performance of the equivalent optimized OpenMP CPU code, compiled with the Intel C++ Compiler, version 11.0, was measured on an Intel Core 2 Q9450 CPU, running either one or four threads.

A NACA0012 wing in supersonic ($M_\infty = 1.2, \alpha = 0^o$) flow was used as a test case. The surface of the mesh is shown in Figure 1. The pressure contours are plotted in Figure 2. Timing measurements when running in single-precision are given in Figure 3 for a variety of meshes, showing an average performance scaling factor of 9.4x in comparison to the OpenMP code running on four cores and 32.6x in comparison to the OpenMP code on one core. Furthermore, the code running on graphics hardware is faster by a factor of 3.9x using redundant computation in comparison to pre-computed flux contributions. Timing measurements when running in double-precision are given in Figure 4 for a variety of meshes, showing an average performance scaling factor of 1.56x in comparison to the OpenMP code running on four cores and 4.7x in comparison to the OpenMP code on one core. Furthermore, the code running on graphics hardware is faster by a factor of 1.1x using redundant computation in comparison to pre-computed flux contributions.

A missile in supersonic ($M_\infty = 1.2, \alpha = 8^o$) flow was used as an additional test case. The pressure contours are plotted in Figure 5. Timing measurements when running in single-precision are given in Figure 7 for a variety of meshes, showing an average performance scaling factor of 9.9x in comparison to the OpenMP code running on four cores and 33.6x in comparison to the OpenMP code on one core. Furthermore, the code running on graphics hardware is faster by a factor 3.4x using redundant computation in comparison to pre-computed flux contributions. Timing measurements when running in double-precision are given in Figure 8 for a variety of meshes, showing an average performance scaling factor of 2.5x in comparison to the OpenMP code running on four cores and 7.4x in comparison to the OpenMP code on one core. Furthermore, the code running on graphics hardware is faster by a factor 1.63x using redundant computation in comparison to pre-computed flux contributions.

V. Conclusions and Outlook

A substantial performance gain has been achieved by using effective techniques which take advantage of the computational resources of modern graphics hardware. Based on these results, it is expected that current and future GPUs will be well-suited and widely used for unstructured grid based solvers. Such an order of magnitude speed-up can result in a significant increase in the scale and complexity of the problems considered in computational fluid dynamics. However, his performance gain is less pronounced in the case of double precision, and it is hoped that future hardware iterations will improve double precision performance.

An open standard, OpenCL,[14] has emerged as an alternative to CUDA. OpenCL is similar to CUDA, and therefore the techniques presented here are expected to be of relevance for codes written using OpenCL.

A more advanced solver is in development which supports fourth order damping, approximate Riemann solvers, flux limiters, and edge-based computation.

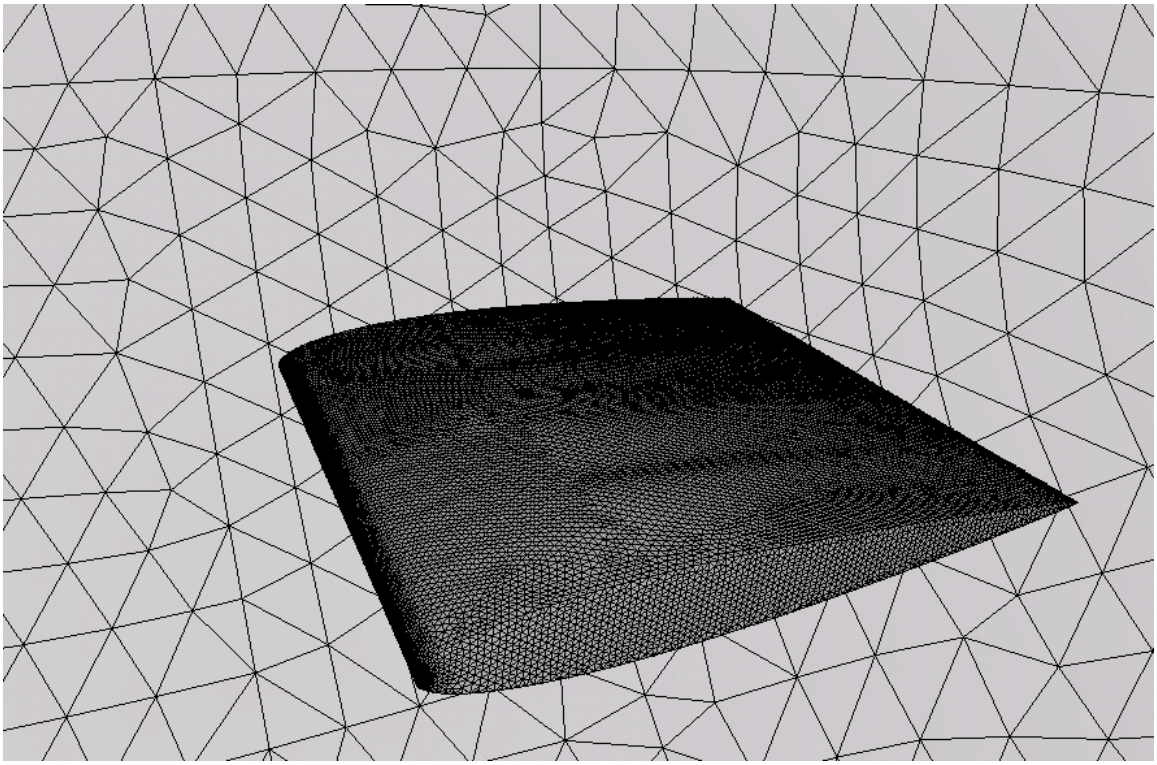

Figure 1. NACA0012 Wing Surface Mesh

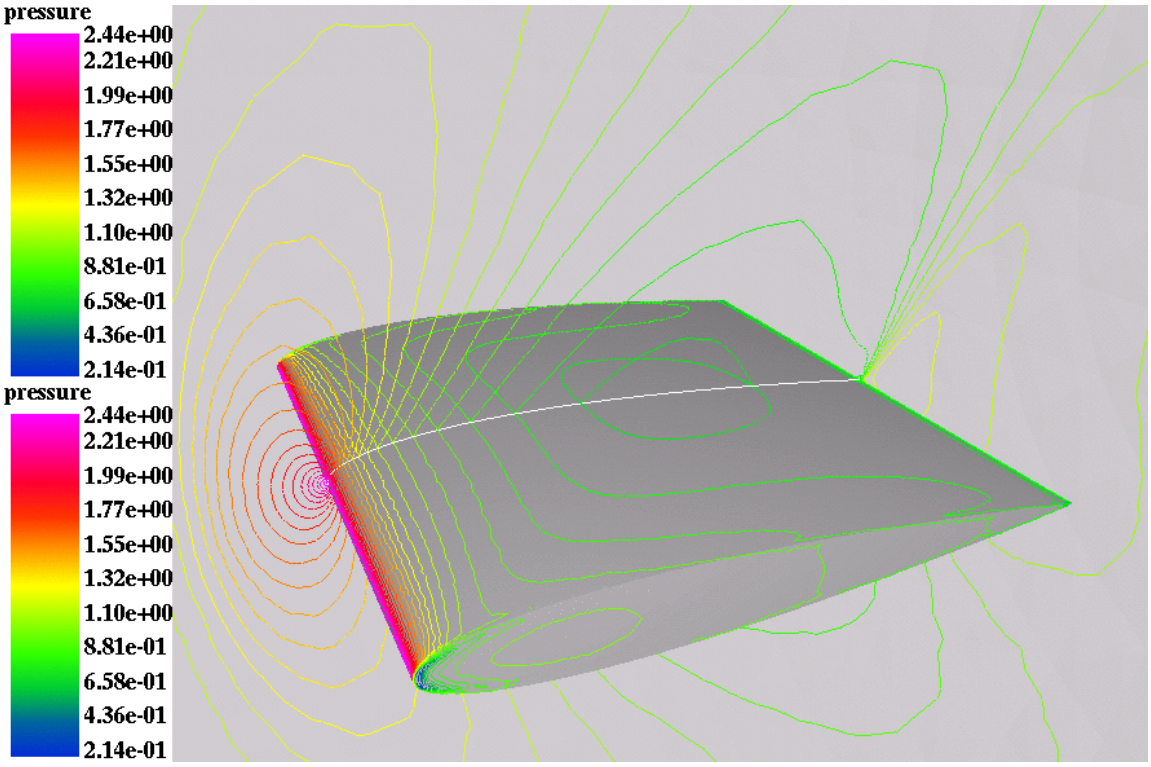

Figure 2. Pressures Obtained at the Surface and Plane for the NACA00012 Wing

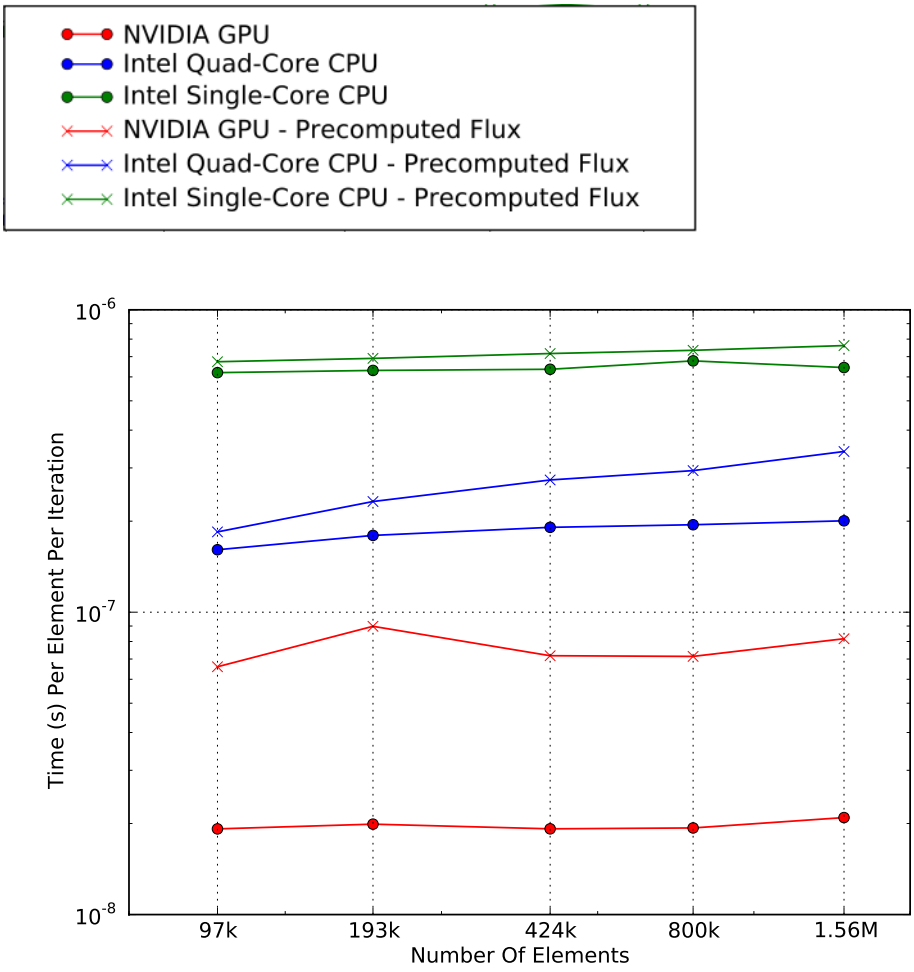

Figure 3. Running Time (s) Per Element Per Iteration for the NACA0012 Wing in Single-Precision.

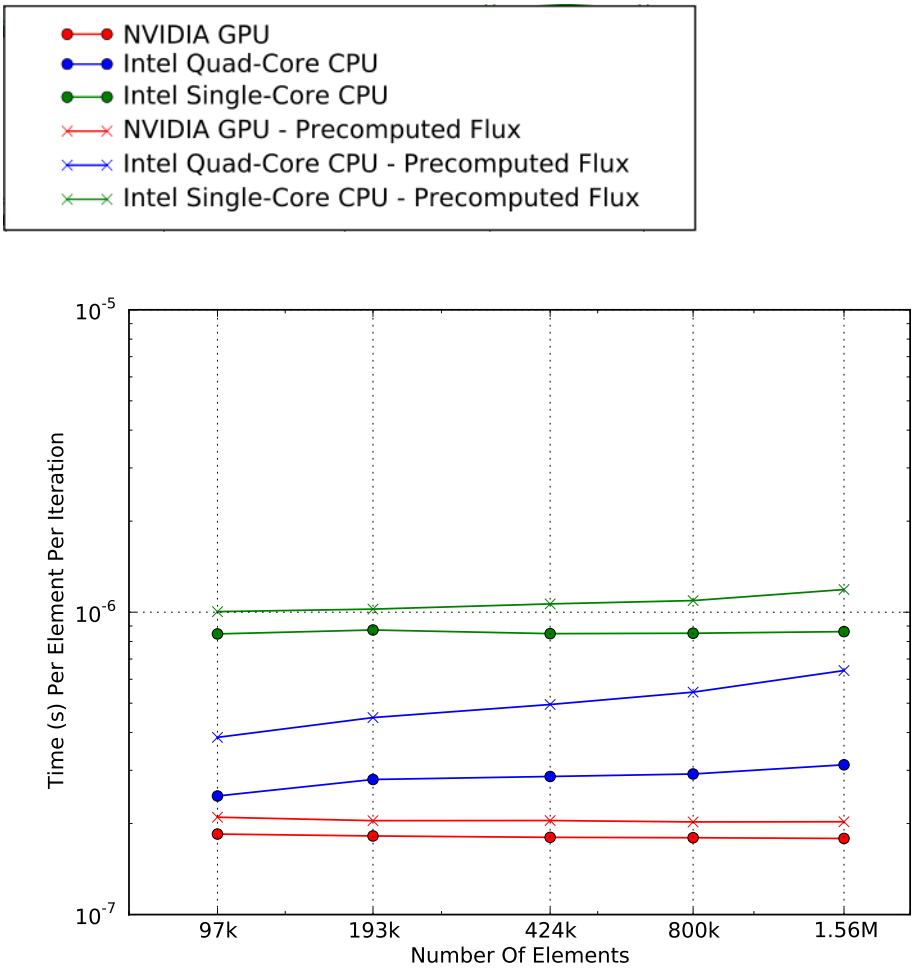

Figure 4. Running Time (s) Per Element Per Iteration for the NACA0012 Wing in Double-Precision.

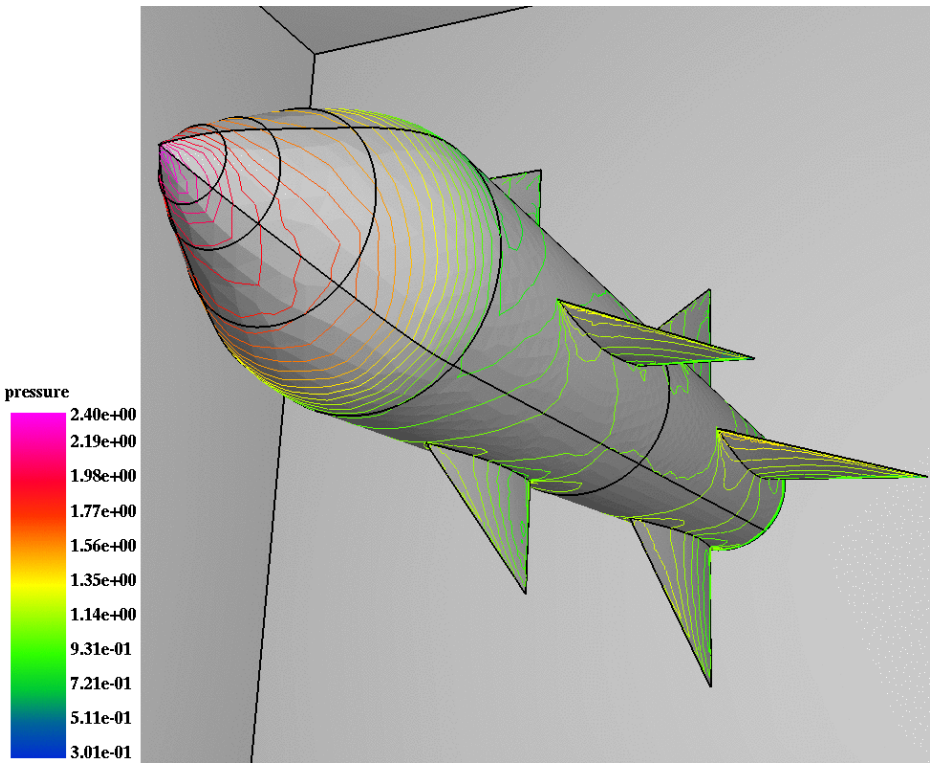

Figure 5. Pressures Obtained at the Surface for the Missile

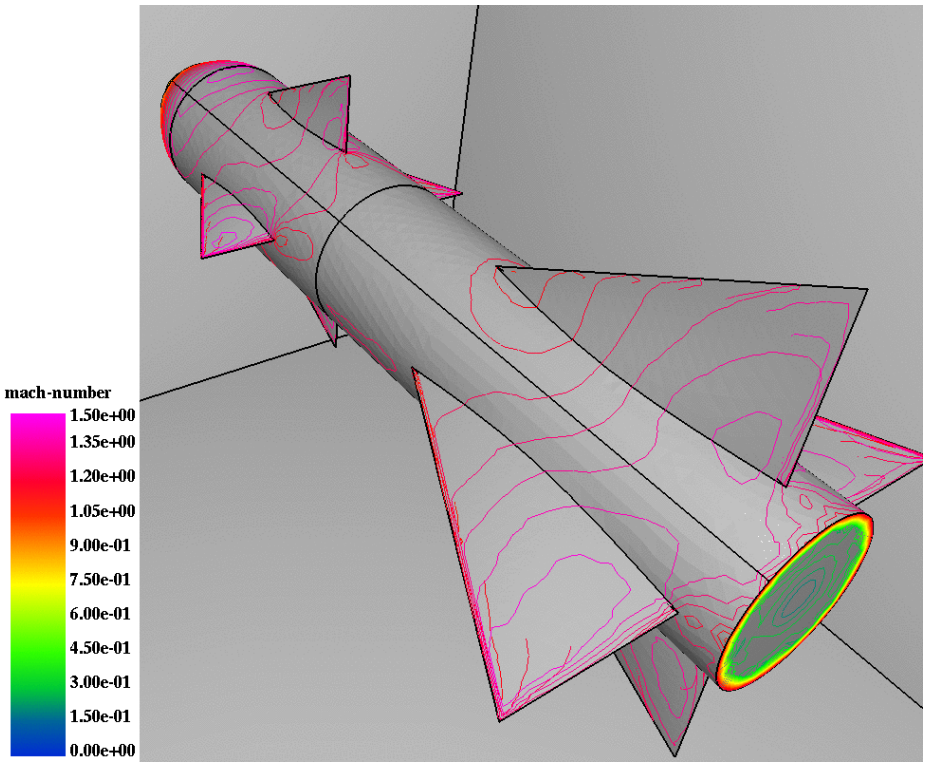

Figure 6. Mach Number Obtained at the Surface for the Missile

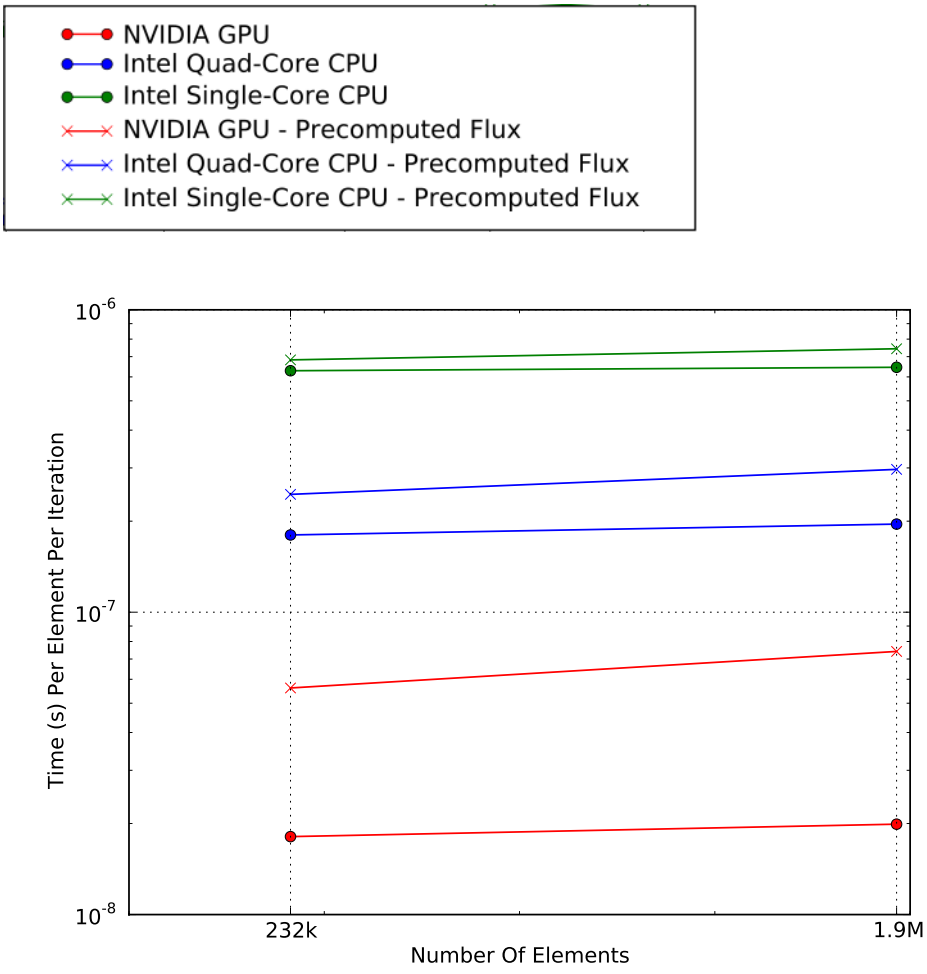

Figure 7. Running Time (s) Per Element Per Iteration for the Missile in Single-Precision.

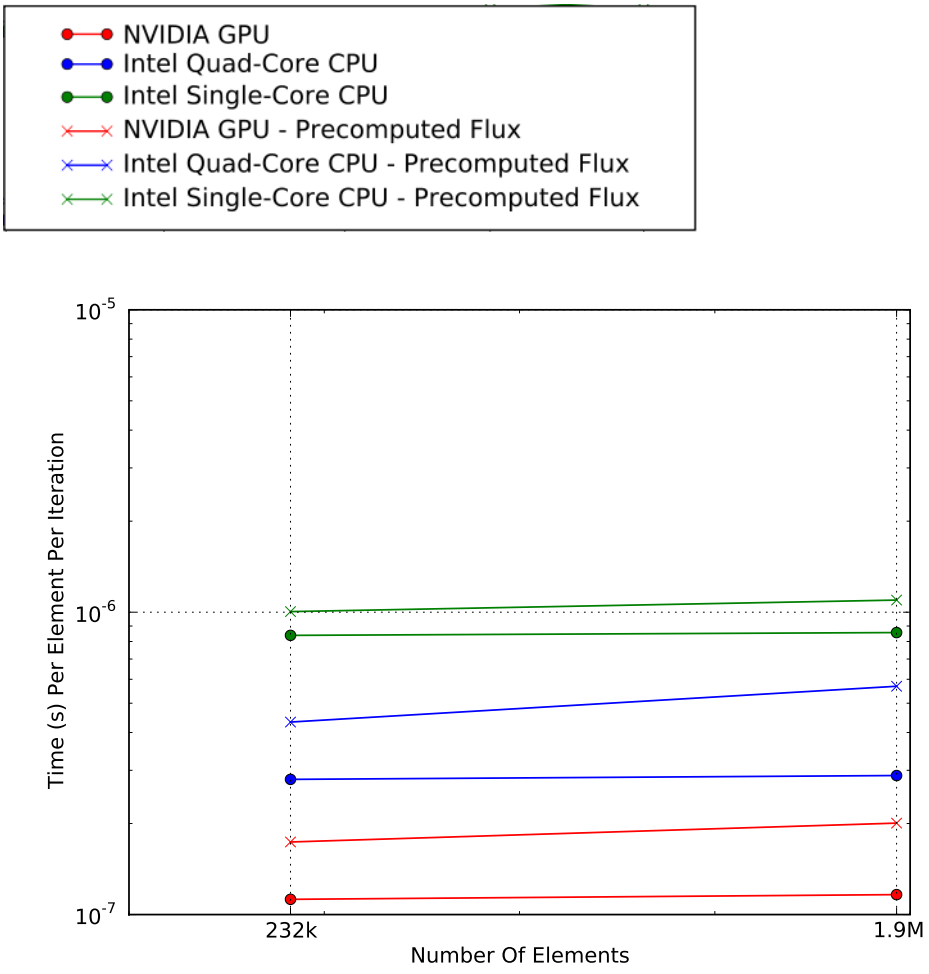

Figure 8. Running Time (s) Per Element Per Iteration for the Missile in Double-Precision.

Acknowledgement

The authors thank Sumit Gupta and NVIDIA Corporation for providing hardware for development and testing.

References

[1]NVIDIA Corporation, *NVIDIA CUDA Compute Unified Device Architecture 2.0 Programming Guide*, 2008.

[2]Brandvik, T. and Pullan, G., "Acceleration of a Two-Dimensional Euler Flow Solver Using Commodity Graphics Hardware," *J. Proc. of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, Vol. 221, 2007, pp. 1745–1748.

[3]Brandvik, T. and Pullan, G., "Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware," *46th AIAA Aerospace Sciences Meeting and Exhibit*, No. AIAA 2008-607, January 2008.

[4]Tölke, J., "Implementation of a Lattice Boltzmann kernel using the Compute Unified Device Architecture developed by nVIDIA," *Computing and Visualization in Science*, 2008.

[5]Phillips, E., Zhang, Y., Davis, R., and Owens, J., "CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows," *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, No. AIAA 2009-565, January 2009.

[6]Thibault1, J. and Senocak, I., "CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows," *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, No. AIAA 2009-758, January 2009.

[7]Klockner, A., Warburton, T., Bridge, J., and Hesthaven, J. S., "Nodal Discontinuous Galerkin Methods on Graphics Processors," 2009, arXiv.org:0901.1024.

[8]Goodnight, N., *CUDA/OpenGL Fluid Simulation*, NVIDIA Corporation, 2007.

[9]Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A. E., and Purcell, T. J., "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, Vol. 26, No. 1, 2007, pp. 80–113.

[10]Harris, M., "Fast Fluid Dynamics Simulation on the GPU," *GPU Gems*, chap. 38, Addison-Wesley, 2004.

[11]C. Scheidegger, J. Comba, R. C., "Practical CFD simulations on the GPU using SMAC." *Computer Graphics Forum*, Vol. 24, 2005, pp. 715–728.

[12]Hagen, T., Lie, K.-A., and Natvig, J., "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.

[13]Löhner, R., *Applied CFD Techniques: An Introduction Based on Finite Element Methods*, Wiley, 2nd ed., 2008.

[14]Khronos OpenCL Working Group, *The OpenCL Specification, Version 1.0*, 2008.