UC Davis

IDAV Publications

Title

Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units

Permalink

<https://escholarship.org/uc/item/0hx199m6>

Authors

Phillips, Everett H.
Zhang, Yao
Davis, Roger L.
et al.

Publication Date

2009

DOI

10.2514/6.2009-565

Peer reviewed

Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units

Everett H. Phillips* Yao Zhang[†] Roger L. Davis[‡] John D. Owens[§]

University of California, Davis, 95616

Researchers have recently used the new programmable capabilities of the Graphics Processing Unit (GPU) to increase the performance of scientific codes. We investigate the use of a cluster of GPUs for large-scale CFD problems and show order-of-magnitude increases in performance and performance-to-price ratio. We implement two separate compressible flow solvers. First, we develop a CUDA-based solver for the 2D compressible Euler equations and verify the results against a reference multi-block code MBFLO. After demonstrating the performance of our Euler solver, we proceed to develop a new version of MBFLO by adding GPU-accelerated subroutines to the existing Fortran codebase. Using an eight-node cluster equipped with 16 NVIDIA 9800GX2 GPUs, we achieve speedups of up to 496x on our Euler Solver and 88x on MBFLO. This paper describes the numerical, hardware and software techniques that provide significant speedups.

I. Introduction

High-performance automated design optimization procedures require an enormous amount of computing power to predict performance with sufficient accuracy to distinguish between configuration alternatives. The cost of using large supercomputers for solving such problems preclude their wide-spread adoption as a design tool. We suggest the use of massively parallel Graphics Processing Units to significantly decrease the cost of large-scale parallelization.

Specifically, we develop a specialized flow solver for the 2D Euler Equations and then use similar techniques to incorporate GPU subroutines into an existing multi-block flow solver, MBFLO¹. The MBFLO code is designed for research in turbulence modeling and turbo machinery design. Our GPU solvers are benchmarked on an eight-node cluster with 32 coupled CPU/GPU processing cores and perform up to 496x faster than our fastest serial CPU implementation. Considering the cost of the GPUs accounts for only 20% of the entire cluster value this translates into an increase in performance-to-price ratio of well over an order in magnitude.

II. Motivation

Current steady Reynolds-averaged Navier-Stokes solution procedures can generally predict aerodynamic performance of various configurations within 3–5% accuracy. The requirement, however, for new high-performance designs that utilize automated optimization procedures is within 0.1% accuracy. Two fundamental bottlenecks make this goal extremely difficult to achieve.

The first bottleneck is the limitation of “steady” simulations, fixed computational grids, and turbulence modeling. The inaccuracy of aerodynamic performance prediction is mostly due to ignoring the effects of self-excited unsteadiness. By solving the Navier-Stokes equations using steady-flow numerical techniques, such as local time-steps and multiple-grid acceleration on fixed computational grids, solution time can be made acceptable for design optimization. However, the saving in solution time is offset by the limited solution accuracy. Recent research has shown that time-averaged, unsteady simulations can be produced with

*Graduate Researcher, Mechanical and Aeronautical Engineering, AIAA Student Member

[†]Graduate Researcher, Electrical and Computer Engineering

[‡]Professor, Mechanical and Aeronautical Engineering, AIAA Associate Fellow

[§]Professor, Electrical and Computer Engineering

multiple CPUs in nearly the same solution time as steady-state simulations using a single CPU. Of course, the cost of doing this becomes exceedingly great when considering complex three-dimensional configurations since the required number of CPUs grows to a very large number. This cost associated with large-scale parallelization is the second bottleneck. If a new technology could be introduced to reduce the cost of large-scale parallelization, time-averaged, unsteady simulations will become routine. Once this occurs, there is little additional cost of using advanced detached-eddy or large-eddy turbulence modeling. So, the second bottleneck of performance-to-price ratio is the most limiting.

The goal of the current effort has been to break these bottlenecks and demonstrate that parallelization performance-to-price ratio can be improved by an order in magnitude by using a coupled CPU/GPU platform.

III. GPU Computing

Figure 1. GPU vs CPU performance trends. GPU performance is growing much faster than the CPU.

While GPUs (Graphics Processing Units) have traditionally been used for real-time rendering applications, the recent addition of programmability has enabled their use in a broad range of general-purpose and scientific computation³, including differential equations for engineering applications^{4–12}. Researchers have also shown that the performance of scientific simulation can be improved by adding GPUs to cluster nodes^{13, 14}.

Figure 1 shows the trends in CPU and GPU performance growth over much of the last decade. The GPU performance increases at a much faster rate than CPUs, providing an order of magnitude higher compute rate at a similar cost. In fact the GPU performance is doubling roughly every six months, compared to CPU performance which doubles about every 18 months. The GPU is also available at a very competitive price point due to mass production to meet the demand for interactive graphics in the video game market.

The architecture of NVIDIA's first Tesla based GPU, the G80, is shown in Figure 2. The hardware consists of an array of 16 SIMD multiprocessors(MPs), each equipped with 8 scalar processing units (SPs), for a total of 128 floating point cores running at 1.35 GHz. In NVIDIA's most recent GPU, the GT200, the number of MPs is increased to 30 for a total of 240 SPs. Each of these scalar processors is capable of executing a maximum of 3 floating point operations per clock cycle which amounts to a peak throughput of 933 GFLOPs. In addition, each SP is able to work on up to 128 concurrent threads, or totally 30,720

Figure 2. G80 GPU Block Diagram (from Owens et al.²). The processor consists of 128 scalar processing units that are capable of executing up to 12,288 concurrent threads.

concurrent threads, which allows the processor to hide latencies with efficient hardware thread scheduling.

We adopt the NVIDIA CUDA programming model¹⁵ in the development of our GPU code because the language is far more mature than anything else available and NVIDIA has committed to supporting the interface for all future products. The CUDA abstraction allows developers to easily write massively parallel programs using extensions to the C language. CUDA makes expressing parallelism very straightforward and can produce very efficient code. Additionally, the latest release of CUDA includes a compiler for use on multi-core CPUs, extending the usability of a CUDA application to systems without GPUs.

IV. 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:

$$\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u_i)}{\partial x_i} = 0$$

Conservation of Momentum:

$$\frac{\partial (\rho u_i)}{\partial t} + \frac{\partial (\rho u_j u_i)}{\partial x_j} + \frac{\partial p}{\partial x_i} = \frac{\partial \tau_{ji}}{\partial x_j} - S \bar{m}_i$$

Conservation of Energy:

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

V. Methodology

We have developed two flow solution procedures. The first version is an Euler procedure developed specifically for the GPU which has limited applicability. The second version consists of the multi-block MBFLO code which is similar to “production” codes used in government and industry with arbitrary block connectivity and orientation. The Euler code provides best-case performance information whereas the MBFLO code provides performance information more typical of existing full-functionality codes.

We solve the compressible flow equations on an irregular structured grid using the finite volume method and a second-order accurate integration scheme developed by Ni¹⁶. Our solution procedures use steady-flow convergence acceleration techniques such as local time-stepping and multi-grid. In MBFLO, the steady solution procedure serves as an explicit inner iteration in a dual time-stepping method.

V.A. Ni’s Distribution Formulae

The distribution formulae suggested by Ni¹⁶ effectively recast the second-order Lax-Wendroff scheme into an accumulation of distributions that occur within each control volume. This reduces the size of the computational stencil and simplifies the treatment of boundary points, especially when used with globally unstructured grids and multi-grid techniques.

The MBFLO solver uses a block connectivity list with each block face divided into an arbitrary number of subfaces. The only current restriction is that blocks must be point matched so boundary points can be updated by accumulating contributions from all neighbors. This restriction can be removed in the future by using interpolation schemes or ghost nodes, but these are beyond the scope of our current work.

In our Euler solver we restrict ourselves to strictly structured grids to ease the initial development but it is possible to add support for globally unstructured grids using similar techniques.

V.B. Numerical Dissipation

Central-differencing numerical methods require artificial dissipation to prevent odd-even decoupling and spurious oscillations near steep gradients. We employ a second-difference smoother to capture shocks, however in regions where gradients are not steep, we use a superior fourth-difference term as suggested by Jameson¹⁷. The smoothing effectively alters the corrections to the dependent variables and is of the form:

$$\Delta U = \alpha \nabla^2 U \left| \frac{\nabla^2 P}{P_{avg}} \right| + \beta \nabla^4 U$$

Here U is the dependent variable, ΔU is the correction variable, α and β are smoothing constants. We turn off the fourth difference smoother when the value of the second difference term becomes larger than the fourth-difference term. This prevents overshoots that typically occur when the fourth-difference term encounters a steep gradient.

V.C. Solution Process Overview

The main algorithm kernels in the procedure consist of:

1. time-step
2. viscous stresses
3. flux integration and distribution
4. numerical dissipation
5. boundary conditions
6. updating of dependent variables
7. application of multiple grid

These steps will be described below in the context of GPUs.

Figure 3. Domain Layout for GPU solution of Euler Equations. Blue: domain; Yellow: row padding; Red: updated region; Brown: compute overlap; Green: smoothing overlap.

VI. Implementation

VI.A. Serial Euler Solver

When designing our Euler solver our main goal was to minimize the use of memory resources. This helps to ensure the solver will continue to scale well on future devices with higher compute-to-memory ratios and also allows for larger subdomains which will help scaling across multiple GPUs in a cluster.

For this reason we chose to only store the primary variables and updated primary variables, as well as the grid point positions. All other variables which may be needed in the computation are computed on-the-fly and stored in the high-speed on-chip shared memory cache.

Each thread-block is responsible for updating a tile of data in the domain as shown in Figure 3. The blocks read the geometry and the latest solution from the tile plus a surrounding area of ghost cells into the shared memory cache. During the reading process we overlap the calculation of nodal variables, such as pressure, enthalpy, and velocity components, with the read since they do not depend on neighbor data. After reading and nodal variable calculations we synchronize the threads of the block.

Next, the secondary geometric variables are computed from the nodal position data, such as areas and volumes of the cells. The shared data is then used to compute the nodal time-steps, integrate fluxes, and apply numerical dissipation. Then, we enforce boundary conditions and update the primary variables.

Testing different block sizes, we find the most optimal for our kernels to be 16×5 tiles of data. We choose the width of 16 because memory access on the GPU is much faster when 16 threads of a half-warps are coalesced into a single memory transaction, and the height of 5 is the largest that can fit within per block register and shared memory limits. Additionally, we pad the rows of each array to a multiple of 16 to meet the coalescing requirements.

After updating the fine-grid, we propagate the current updates on progressively coarser grids using the distribution formulae once again. Finally, we interpolate the coarse corrections and add them to the solution.

VI.B. Parallel Euler Solver

We implement our Parallel Euler solver on an eight node cluster with a total of 32 GPU cores. We divide the domain into multiple subdomains and assign each to a GPU. Figure 4 shows a domain divided into four pieces for a four-GPU implementation. We overlap the neighboring pieces by two lines of ghost nodes because the fourth-difference smoothing requires two neighbors.

For parallelization the local GPUs execute the same code developed for the single GPU solver with the addition of communication oriented kernels that work with OpenMPI to transfer data between subdomains. We use a Master-Worker process relationship with each worker having a single GPU. The master divides the domain, calculates initial conditions, and sends work to the workers. Each of the worker CPUs then copy their subdomain to the GPUs and begin a series of kernel launches and MPI calls to advance the solution. After the workers update their local domains for a given time-step, a kernel gathers together all data for communication into a pre-allocated send-buffer in GPU memory. The worker then copies the contents of the buffer to main system memory and calls MPI routines to exchange information with neighboring nodes. After communication, the worker copies the received buffers from system memory to the GPU receive buffer

Figure 4. A simple 1-D domain decomposition across multiple GPUs in the cluster. Each subdomain communicates two lines of data with neighbors for smoothing purposes.

and executes a kernel that scatters the data in the buffer to the corresponding portions of the GPU arrays. The gathering and scattering of the data allows the memory copies to be processed more efficiently as a single large memory copy is usually more efficient than many smaller ones. After convergence of the solution, the worker processes send their partial results to the master which assembles them and writes the result to disk.

VI.C. MBFLO

MBFLO is a Fortran based multi-block flow solver for compressible flow with options for inviscid, laminar, turbulent RANS, and advanced turbulence modeling approaches. We focus on the 2D version of the code which can solve both planar and axi-symmetric configurations. This version contains approximately 100 subroutines and roughly 40,000 lines of Fortran. With such a large code, we currently choose to only accelerate the most commonly used functions that form the basis of the entire program. These are the unsteady flow and laminar stress routines, which are also used in turbulent simulations. We did not accelerate the boundary conditions or block-to-block communication routines, as these represent a small fraction of the total compute time and would require a considerable amount of effort.

We also made the decision early on to only optimize the code for the newer architecture of the GT200. The reason for this was to reduce the time for development. The GT200 has a much more flexible memory system, has larger register files, and also supports double precision. Although we only have a single GT200 card, we plan to outfit our cluster with these cards in the near future.

We start by profiling the existing code and begin our work on the flux, smoothing, and deltat routines which take the largest portion of the processing time. Initially, we handle all GPU memory management in the individual GPU subroutines in order to reduce the amount of code changes in the early stages of kernel development and debugging. After a few important subroutines are verified to function correctly, we then move the GPU memory management into the Fortran code, and leave only the computational kernels themselves in the subroutines.

We then proceed to profile the new partially accelerated code and continue to gradually add additional routines. During the entire process, although only a portion of the code is moved to the GPU, the full functionality of the original program is retained.

In our initial implementation we copy the entire solution from the GPU to CPU before and after the block boundary treatment. This does not affect performance much when only a few functions are accelerated. However, after moving all of our routines to the GPU, the memory copies were responsible for 50% of the total compute time. We remedy this by creating a set of functions that gather the entire boundary and ghost node data into a buffer, and only transfer this one variable to the CPU. After the block boundary routine is finished we pack the boundaries from the CPU arrays to another buffer, send this to the GPU, and use another kernel to unpack and scatter the contents to the boundaries of the GPU arrays. This resulted in a two fold increase in performance on the GT200.

Currently we have added GPU versions of all the unsteady and laminar stress portions of the code. In addition we create functions for GPU memory management, kernel debugging and data structure optimizations. The memory functions allocate, deallocate, and manage transfers to and from GPU memory. The debugging routine compares the CPU and GPU output of subroutines. The data optimization routines convert GPU array layouts between arrays of vectors and vectors of arrays to increase memory bandwidth.

The total cost of development involved the addition of 500 lines of Fortran, 2500 lines of CUDA, and approximately three man-months of effort by experienced CUDA programmers.

VII. Results

VII.A. Test Cases

Figure 5. Results of GPU Euler solver showing pressure contours and grid geometry in a nozzle and over a supersonic diamond airfoil. The Airfoil case demonstrates the shock capturing capability of the algorithm.

Figure 6. Results of MBFLO showing entropy contours in an unsteady compressible laminar flow over a cylinder at Mach Number 0.10 and Reynolds number 140.

We test our GPU Euler code on both a subsonic internal flow through a nozzle at Mach Number 0.3 and a supersonic external flow over a diamond airfoil at Mach Number 2.5. The steady state pressure distributions are plotted in Figure 5. We test MBFLO with GPU acceleration on the flow over a cylinder at a Reynolds number of 140 and Mach number of 0.1. The entropy contours at non-dimensional time 200 are shown in Figure 6. Each test case was verified to agree with our MBFLO reference CPU results.

VII.B. Serial Performance

The serial version of our Euler code was tested on various CPU architectures as well as the NVIDIA 8800GTX GPU and the results are given in Figure 7. The performance is measured by dividing the total number of

Figure 7. Serial Euler solver performance for nozzle test case of varying grid size.

grid points by the time for a single iteration of the solver yielding the throughput in millions of mesh points per second. For the smallest grid tested the level of parallelism is relatively small and the GPU is only able to outperform the fastest CPU by 3.5x. As the mesh size is increased the GPU becomes more efficient and the performance grows to over 20x that of any of the CPUs. In contrast the CPU performance decreases as the problem size grows larger than the size of the L2 cache.

The serial MBFLO code performance has incrementally increased as more of the code is moved onto the GPU. The progress on some of the key computational kernels relative to a single core of the 2.5 GHz Q9300 CPU is shown in Figure 8 for both G92 and GT200 class GPUs. The final speedup for the GT200 and G92 are 14x and 6.8x respectively.

VII.C. Cluster Performance

The parallel versions of our codes are tested on an eight-node cluster with each of the nodes equipped with NVIDIA 790i Ultra motherboards, dual 9800GX2 graphics cards, an Intel Q9300 Quad-Core Processor, and

Figure 8. MBFLO acceleration of the main Timestep routines is done incrementally by adding GPU support to more of the subroutines and optimizing data movement between CPU and GPU. The above timing results are obtained on a steady laminar flow cylinder test case with a mesh size of 1025×257 .

Figure 9. Parallel Euler solver performance for varying mesh size Nozzle test cases relative to a single Core2Duo CPU. Multiple GPU performance benefits are not realized until the domain reaches 1537×257 grid points. As the domain size is increased the code scales better, reaching a maximum speedup of 496x with 32 GPUs on the 6145×1025 domain.

Figure 10. Parallel MBFLO solver performance for varying mesh size Unsteady Laminar Cylinder test cases. 16 GPUs perform up to 88x faster than a single CPU and roughly 6x faster than 16 CPUs.

8 Gigabytes of DDR3 memory. The cluster runs the open source ROCKS operating system and uses gigabit Ethernet for management and message passing.

We use the nozzle test case to benchmark the parallel Euler code on GPU clusters of varying size. The resulting performance, scaled to a single core of the Q9300 2.5 GHz CPU, is shown in Figure 9. For the smallest domain size of 97×17 , there is not enough work to keep even a single GPU busy and performance only degrades when adding additional processors. In the 385×65 case, the addition of multiple GPUs is only marginally better than a single GPU. However, as the domain size is further increased, the amount of work in each subdomain becomes enough to offset the communication costs and the cluster performance increases with additional GPUs. The maximum performance of 496x is reached with 32 GPUs on the largest domain of 6145×1025 points. This trend is likely to continue, with better scaling as the domain size is increased, and greater performance with additional GPUs. The scaling can also be improved if a high-speed interconnect was used, although this would add additional cost to the cluster.

The MBFLO code with and without GPU acceleration is benchmarked on the Unsteady Laminar Cylinder test case with both 1025×769 and 2049×1537 mesh sizes. The resulting throughput in mesh points per second relative to the single CPU is shown in Figure 10. Similar to the Euler code, as domain size is increased, the parallel efficiency increases and the performance improves when adding additional GPUs. An 88x speedup on 16 GPUs for the 2049×1537 case is observed, which is 6x faster than the same number of CPUs. The code performs relatively well considering it has not been optimized for the G92 GPUs that are used in the test.

VIII. Discussion

When developing the GPU routines we noticed very high performance relative to the CPU code, even without using advanced optimization techniques. In fact, in our Euler code, the main computational kernel uses over 40 registers which reduces the number of concurrent threads that the GPU can keep in flight simultaneously. Therefore it may be possible to improve the performance well beyond what we have demonstrated. Again, our goal is to show practical performance improvement without spending a great deal of development time tuning the code.

In our experience the GT200 requires substantially less effort to create high performance code. The addition of a few key hardware features makes the GT200 much better suited to scientific computation: double-precision floating point support, doubling of the number of registers on each multiprocessor, and relaxed memory coalescing requirements.

Further performance increases of our solvers would involve more experimentation. One approach is to split the large kernels into several smaller ones. It is not clear whether or not this will increase performance because as the kernel size is reduced, allowing higher occupancy, the number of memory transactions is increased to store the intermediate results. We could also use asynchronous streaming features to overlap communication with computation to increase the parallel efficiency.

CFD codes are an excellent fit for data parallel architectures. Developing a flow solver that takes advantage of new high-performance accelerators requires a basic understanding of the architecture and some experimentation. Starting from scratch, as we did with our Euler routine, makes it easier to optimize for performance because the kernel organization, data structures, and program flow can be designed with the most flexibility. In the long term redesigning an existing code will provide maximum performance. However, with very little effort, substantial performance gains can be made by simply augmenting existing code with GPU acceleration. There is a large base of existing CFD and other scientific discipline codes used in practice that could benefit greatly from performance enhancement with GPU processors.

IX. Conclusions

Currently we have demonstrated the use of a GPU cluster for increasing the performance-to-price ratio of compressible flow calculations. With an eight-node cluster of 32 GPUs, our custom Euler solver achieves up to a 496x speedup over a single CPU. We also extend the capabilities of the existing MBFLO solver and achieve incremental speedups as more of the code is moved onto the GPU and optimized for the GPU architecture. With only 16 GPUs we accelerate the code up to 88x over a single CPU or 6x over 16 CPUs.

In the future we expect to improve both the performance and capability of our solvers with extension to turbulent flows and 3D problem domains. Interesting directions for future research include the application

of GPUs to adaptive mesh refinement, and the development of open source software tools for acceleration of CFD applications.

X. Acknowledgments

The authors appreciate the sponsorship of this work by the Department of Energy's Early Career Principal Investigator Award (DE-FG02-04ER25609) and the managers of Wright-Patterson Air Force Research Laboratory under contract 03-S530-0002-11-C1 with Dr. John Clark as technical monitor. Thanks also to NVIDIA for hardware donations.

References

- ¹Davis, R. L. and Dannenhoffer III, J. F., "A Detached-Eddy Simulation Procedure Targeted for Design," Vol. 24, No. 6, Nov./Dec. 2008.
- ²Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C., "GPU Computing," *Proceedings of the IEEE*, Vol. 96, No. 5, May 2008.
- ³Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, 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. 80–113.
- ⁴Bolz, J., Farmer, I., Grinspun, E., and Schröder, P., "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Transactions on Graphics*, Vol. 22, No. 3, July 2003, pp. 917–924.
- ⁵Brandvik, 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.
- ⁶Goodnight, 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.
- ⁷Hagen, T. R., Hjelmervik, J. M., Lie, K.-A., Natvig, J. R., and Henriksen, M. O., "Visual simulation of shallow-water waves," *Simulation Modelling Practice and Theory*, Vol. 13, No. 8, Nov. 2005, pp. 716–726.
- ⁸Harris, 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.
- ⁹Krüger, 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.
- ¹⁰Li, W., Fan, Z., Wei, X., and Kaufman, A., "GPU-Based Flow Simulation with Complex Boundaries," *GPU Gems 2*, edited by M. Pharr, chap. 47, Addison Wesley, March 2005, pp. 747–764.
- ¹¹Scheidegger, 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.
- ¹²Hagen, 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.
- ¹³Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S., "GPU Cluster for High Performance Computing," *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, 2004, p. 47.
- ¹⁴Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S. H., Grajewski, M., and Turek, S., "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster," *Parallel Computing*, Vol. 33, No. 10–11, 2007, pp. 685–699.
- ¹⁵NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide," <http://developer.nvidia.com/cuda>.
- ¹⁶Ni, R.-H., "A multiple grid scheme for solving the Euler equations," *5th Computational Fluid Dynamics Conference*, June 1981, pp. 257–264.
- ¹⁷Jameson, A., Schmidt, W., and Turkel, E., "Numerical Solutions of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time-Stepping Schemes," *14th Fluid and Plasma Dynamics Conference*, No. AIAA-1981-1259, June 1981.