Title

Acceleration of 2-D Compressible Flow Solvers with Graphics Processing Unit Clusters

Journal

Journal of Aerospace Computing Information and Communication, 8(8)

ISSN

1542-9423

Authors

Phillips, Everett H

Zhang, Yao

Davis, Roger L

et al.

Publication Date

2011-08-01

DOI

10.2514/1.44909

Acceleration of 2D Compressible Flow Solvers with GPU Clusters

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

Department of {MAE, ECE}, One Shields Avenue, University of California, Davis, 95616

We demonstrate the first use of GPU clusters for engineering-level compressible flow computations with two separately developed solvers. Each solver adopts the finite-volume approach on multi-block, structured, non-uniform, quadrilateral meshes. We use a second-order accurate integration scheme, with a second-order accurate dual-time integration, using steady-flow acceleration techniques such as local time-stepping and non-linear multi-grid. The first solver is based on an existing multi-block Fortran code, MBFLO, which is augmented with GPU acceleration for the laminar and unsteady flow subroutines. The second solver is built from scratch for GPU acceleration, but is limited to the steady Euler solutions over simple geometries. The Euler solver is optimized by reducing the memory bandwidth, recomputing intermediate values and combining functions to increase producer-consumer locality. These two approaches are used to highlight trade-offs between development effort and performance when accelerating an application with GPUs. On test cases with up to 6.4 million grid points, our GPU solvers outperform their CPU counterparts by up to 20x. In our GPU strong scalability test, 32 GPUs scale up to 22x over a single GPU, which is 16x faster than 32 CPU cores, or 500x faster than a single CPU core. It is confirmed that GPU clusters are effective for engineering CFD applications if their order-of-magnitude increase in performance is paired with large problem sizes to amortize the cost of communication.

*Graduate Researcher, Mechanical and Aeronautical Engineering, AIAA Student Member
†Graduate Researcher, Electrical and Computer Engineering
‡Professor, Mechanical and Aeronautical Engineering, AIAA Associate Fellow
§Associate Professor, Electrical and Computer Engineering

I. Introduction

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 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.

We suggest the use of massively parallel Graphics Processing Units to significantly decrease the cost of large-scale parallelization.

II. GPU Computing

The massively parallel graphics processor has recently gained a great deal of attention from the scientific community as the architectures have evolved into general purpose processors capable of accelerating a wide range of non-graphics computations.[1] The interest in GPUs is largely due to the high floating point throughput compared to traditional CPU architectures. As shown in Figure 1, the GPU platforms have clearly outpaced CPUs over much of the last decade, and this trend is likely to continue. The GPU is also available at a very low price point due to mass production to meet the demand for interactive graphics in the video game market.

One can appreciate the level of parallelism in a modern GPU by observing the architecture of the G80, shown in Figure 2. The graphics chip 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 the GT200 GPU, 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 30,720 concurrent threads in aggregate, which allows the processor to hide memory latency with efficient hardware thread scheduling.

The choice of GPUs for the current effort was largely influenced by the ease of programming using the Compute Unified Device Architecture (CUDA)[2] programming model. The abstraction is based around the C programming language with minimal extensions added to support the integration of GPUs as massively parallel co-processors. Function qualifiers are used to mark subroutines for execution on the GPU (device). The CPU (host) code is responsible for allocating memory on the device, and copying data to and from the device. Figure 3 shows the basic structure of a CUDA program. The model is based on a heterogeneous computing model, with serial portions of the application continuing to execute on the CPU, while the highly parallel compute intensive portions are moved to the GPU. Researchers have recently ported CUDA for use on multi-core CPU platforms[3] which allows CUDA applications to run efficiently on both CPUs and GPUs.

III. Previous Work

As mentioned above, the use of programmable graphics processors to accelerate non-graphics tasks (GPGPU)[1] 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,[4–8] which are interesting from a performance and simulatoin 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.[9] begin to incorporate more advanced numerical methods such as high-resolution finite volume with runga-kutta time integrations, and begin to 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 come about in the work of Brandvik and Pullan[10] who solve both 2D and 3D Euler equations on single block non-uniform meshes. 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.[11] 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.

We build on this work, taking a similar multi-block approach, but add support for cluster level computing with multiple GPUs. We also employ a more complex multi-grid method, and add support for viscous 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.

Previous work with GPU Clusters have proven succesful in other fields. Fan et al.[12] use 30 GPUs to simulate the dispersion of airborne contaminants using the lattice Boltzmann model (LBM). They observe a speedup of 4.6x and discuss other GPU Cluster applications. The most recent work on GPU Clusters is that of Göddeke et al.[13] who use the GPU to accelerate a Solid-Mechanics code using a "minimally invasive integration" of GPUs as co-processors. The experiments performed demonstrate scalability on up to 160 nodes with minimal changes made to application code. They remind us that Amdahl's Law is a an important consideration when adding parallelism: since only 2/3 of their code was accelerated, the order of magnitude speedup provided by the GPUs resulted in 2–3x overall improvement.

IV. Methodology

A. Overview

We have developed two flow solution procedures. The fist 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 second version is an Euler procedure developed specifically for the GPU, however this version has limited applicability since it only supports a few boundary condition types and can only deal with simple geometries with 1D block connectivity. Thus, the Euler code provides best-case performance information whereas the MBFLO code provides performance information more typical of existing full-functionality codes. To our knowlege the current effort is the first to address GPU acceleration of compressible flow solvers on distributed computing platforms.

B. Governing Equations and Numerical Approach

We solve the compressible flow equations on a non-uniform structured grid using the finite volume method and a second-order accurate integration scheme developed by Ni.[14] 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 for unsteady flow simulations.

The input to our solvers are grid geometry, boundary conditions, block connectivity, and various options which control the solver behavior: use CPUs or GPUs, axi-symmetric, viscous terms or turbulence models, smoothing parameters, time-step options, reference flow conditions, etc. The output is a file contianing

computed values of the flow solution at each location in the mesh. The current GPU accelerated routines solve the following governing equations for compressible laminar flow:

Conservation of Mass:

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

Conservation of Momentum:

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

Conservation of Energy:

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

C. Distribution Formulae

The distribution formulae suggested by Ni[14] effectively recast the second-order Lax-Wendrof 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.

D. 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 Jameson.[15] 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 large. This prevents overshoots that typically occur when the fourth-difference term encounters a steep gradient. We decay the fourth difference smoothing term in viscous flow regions so that the flow gradients do not get diffused.

E. Solution Process

The main algorithm kernels in the procedure consist of:

1. preprocessing on CPU
2. setup and transfer inital data to GPU
3. begin time loop on GPU
   (a) time-step
   (b) viscous stresses
   (c) flux integration and distribution
   (d) numerical dissipation
4. transfer boundary to CPU
5. boundary conditions (on CPU)
6. transfer boundary to GPU
   (a) updating of dependent variables
   (b) application of multiple grid
7. end time loop
8. transfer solution to CPU
9. post processing on CPU

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

V. Implementation

A. Serial Euler Solver

When designing our Euler solver, our main goal was to maximize performance on the GPU. Previous work indicates that memory bandwidth is a bottleneck for most Euler implimentations, and considering the GPU has an even higher compute-to-memory ratio than CPUs, it makes sense to reduce memory bandwidth, even at the expense of adding additional compute opperations. We took this approach to the extreme, storing only the absolutely nessasary data, and ensuring our solver would make a minimum number of passes over the data during the solution process. This also has a side effect of reducing the memory footprint of the solver, which helps accomodate larger problems. Since communication costs are proportional to the domain surface area, while compute is proportional to the domain volume, larger problems should help increase the overall scalability on the cluster platform. We also note that compute performance generally grows much faster than memory performance, so it is likely that future devices will have even larger compute-to-memory ratios, in which case our approach will have maximum benefit.

For this reason, we chose to only store the primary variables and updated primary variables, as well as the grid point positions. All other quantities that 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 4. 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 18×7 thread-blocks, which update 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-warp 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 during the multi-grid cycle. Finally, we interpolate the coarse corrections and add them to the solution.

B. Parallel Euler Solver

We implement our parallel Euler solver on GPU clusters with a total of 32 GPU cores. We divide the domain into multiple sub-domains and assign each to a GPU. Figure 5 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 MPI to transfer data between sub-domains.

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 copies their sub-domain 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.

Next, the worker copies the received buffers from system memory to the GPU receive buffer 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.

C. MBFLO

MBFLO[16] is a Fortran based multi-block compressible flow solver with options for inviscid, laminar, turbulent RANS, and advanced turbulence modeling approaches. We focus on the 2D version of the code that 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 choose to only accelerate the most commonly used functions that form the basis of the entire program. These are the flux, numerical dissipation, timestep, laminar stress, and unsteady flow 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 memory system behavior of the GT200. The GT200 automatically coalesces memory transactions which allows for more streamlined code, since fewer instructions are required to manually enforce optimal memory access patterns. This also makes the code less complex and easier to read as well. However, on older hardware that does not include

this feature, performance will be substantially degraded because memory accesses that are not coalesced are serialized.

Recalling Amdahl's law, we decided to accelerate the most time consuming routines first: flux, smoothing, and timestep calculation. We continually profiled the partially accelerated code and gradually added additional routines. During the entire process, although only a portion of the code was moved to the GPU, the full functionality of the original program was retained.

In our initial implementation, we copy the entire solution from the GPU to the 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. Similar to our parallel Euler code, we created gpu subroutines to consolodate all of the boundary data and minimize both the size and number of PCIe transfers, resuting in a two-fold speedup.

The performance of our solvers is now limited by the block boundary routines which communicate boundary data between neighboring blocks. We suspect this time can be reduced by eliminating additional CPU memory traffic by packing the boundary data into contiguous chuncks corresponding to individual MPI send and recieve buffers. If there were hardware support for direct communication between the GPU and the network we could avoid the additional copy latencies and skip the CPU altogether, transfering directly from GPU to GPU.

VI. Results

A. Test Cases

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 6. We also test MBFLO with GPU acceleration for laminar viscous 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 7.

B. Serial Performance

The serial version of our Euler code was tested on various CPU architectures as well as the G80-based 8800GTX GPU and the results are given in Figure 8. The performance is measured by dividing the total number of 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 test case, the level of parallelism does not fully utilize the massively parallel GPU architecture, and performance is only 3.5x above that of the CPU. However, 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 throughput decreases as the data exceeds the size of the L2 cache.

The serial MBFLO code was incrementally accelerated as more of the code was 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 9 for both G92 and GT200 class GPUs. The final speedup for the GT200 and G92 are 14x and 6.8x respectively.

C. Cluster Performance

The parallel versions of our codes are tested on two clusters each equiped with at least 32 GPUs. The first cluster has 8 nodes, each equipped with: two 9800GX2 dual GPU graphics cards, a Q9300 Quad-Core Processor, and 8 GB of 1333 MHz DDR3 memory. The second cluster is located at Argonne National Labs. This cluster is equiped with 24 nodes, each containing 2 Tesla C1060 GPUs, dual Quad-Core Xeon E5430 CPUs, and 16 GB of ECC 667 MHz DDR2 memory. Both clusters use the gigabit ethernet interconnect for 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 CPU, is shown in Figure 10. The results indicate that the scalability is significantly affected by the local domain size. As domain size is increased, the ratio of computation to communication improves and performance approches the ideal 100% parallel scalability. It is also important to note that the GPU cluster is not suitable for small problems with less than several hundred thousand grid points.

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 11. Similar to the Euler code, as domain size is increased, the parallel efficiency increases and the performance improves when adding additional GPUs. We observe a 146x speedup on 32 Tesla C1060s for the 2049×1537 case, which is 8x faster than 32 CPUs.

VII. Discussion

When developing the GPU routines we noticed very high performance relative to the CPU code, even without using advanced optimization techniques. Our current implementation is very straightforward, and as a result, some of our main computational kernels consume a large number of registers, which limits performance by decreasing the number of threads that the GPU can process simultaneously.

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 memory transfers to overlap communication with computation and further increase the parallel efficiency.

In our experience, the GT200 requires substantially less effort to create high performance code since fewer instructions are required to ensure that memory accesses are coalesced into 16 adjacent and aligned address locations. The GT200's native support for double precision shows promise for mixed precision methods which may be required when moving to higher Reynolds number flows with fine wall spacing.

Explicit and point-implicit CFD codes are an excellent fit for data parallel architectures. 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. Thus, 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 codes used in practice that could benefit greatly from performance enhancement with GPU processors.

VIII. Conclusions

We have demonstrated the use of GPU clusters for engineering compressible flow solvers and show order of magnitude speedups in performance and performance-to-price ratio over traditional hardware. With continued advances in performance and programmability, commodity data-parallel architectures, like the GPU, have the power to reshape the design process.

In the future we expect to improve the capability of our solvers with extensions to turbulent flows and 3D problem domains. Interesting directions for future research include the application of GPUs to automated design optimization, and the development of software tools for acceleration of CFD applications.

References

[1]Owens, 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.

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

[3]Stratton, J. A., Stone, S. S., and mei W. Hwu, W., "MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores," Tech. Rep. IMPACT-08-01, University of Illinois at Urbana-Champaign, Center for Reliable and High-Performance Computing, April 2008.

[4]Bolz, 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.

[5]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.

[6]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.

[7]Krüger, 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.

[8]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.

[9]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.

[10]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.

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

[12]Fan, 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.

[13]Göddeke, D., Wobker, H., Strzodka, R., Mohd-Yusof, J., McCormick, P., and Turek, S., "Co-Processor Acceleration of an Unmodified Parallel Solid Mechanics Code with FEASTGPU," *International Journal of Computational Science and Engineering (IJCSE)*, 2008, to appear.

[14]Ni, R.-H., "A multiple grid scheme for solving the Euler equations," *5th Computational Fluid Dynamics Conference*, June 1981, pp. 257–264.

[15]Jameson, 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.

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

[17]Owens, 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.

IX. List of Figures

X. Figures

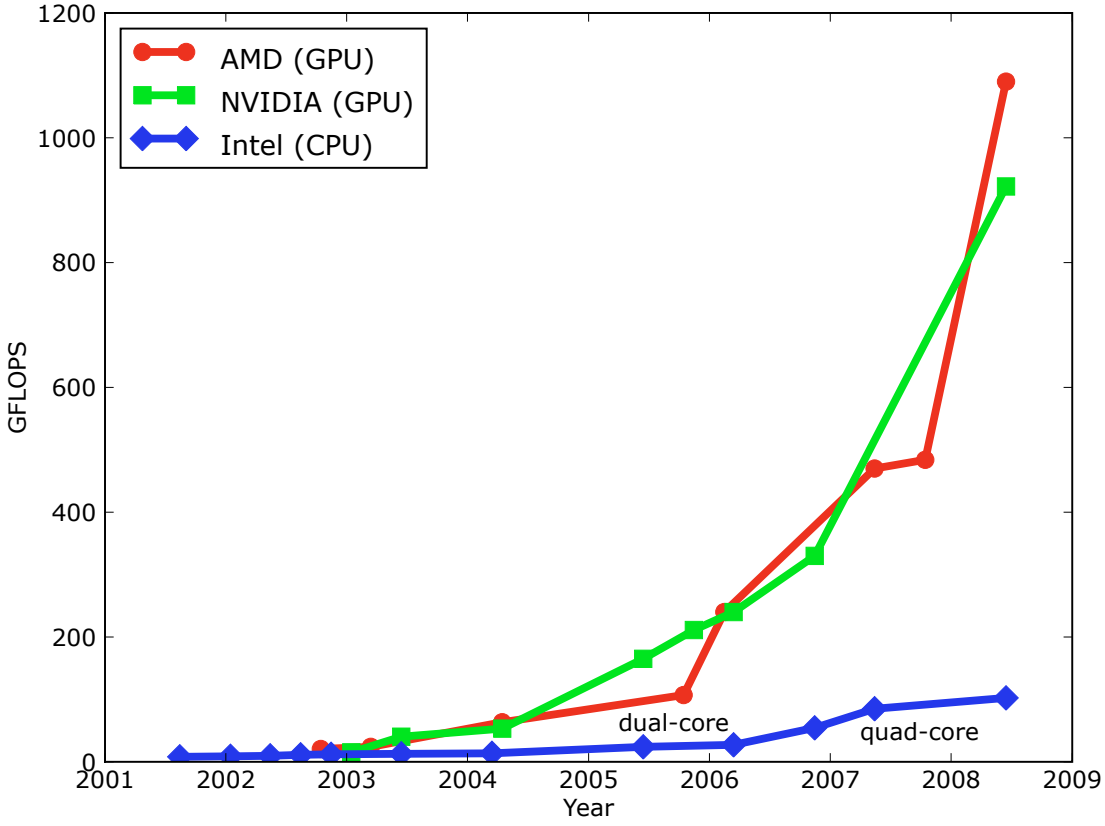

Figure 1. GPU vs CPU performance trends. GPU performance is growing much faster than the CPU.

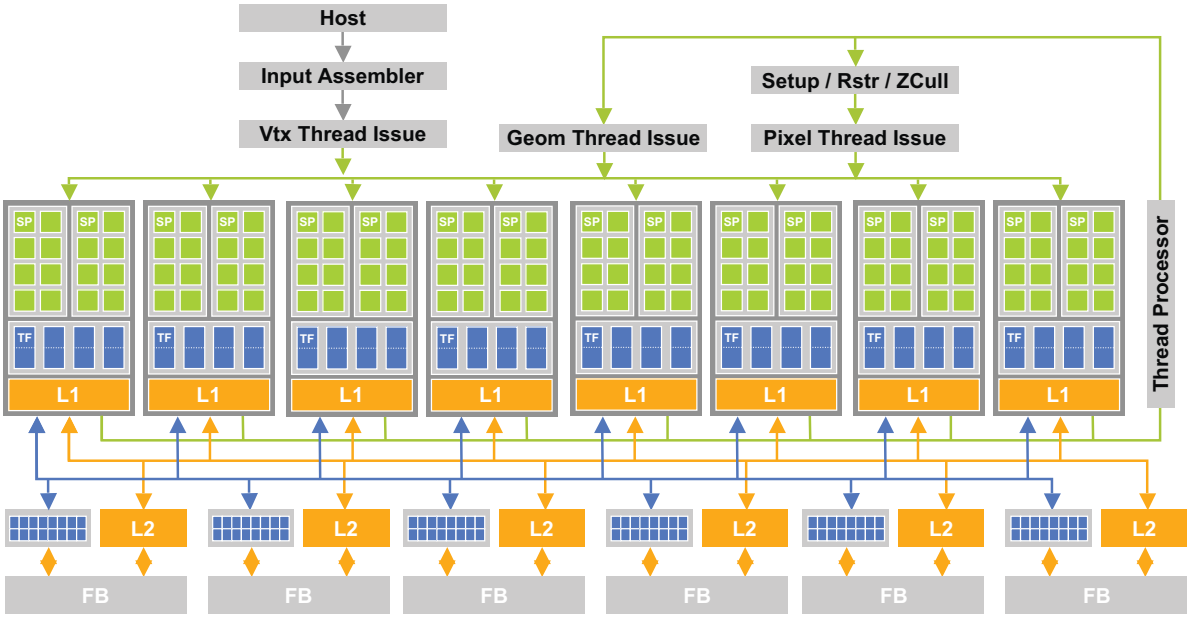

Figure 2. G80 GPU Block Diagram (from Owens et al.[17]). The processor consists of 128 scalar processing units that are capable of executing up to 12,288 concurrent threads.

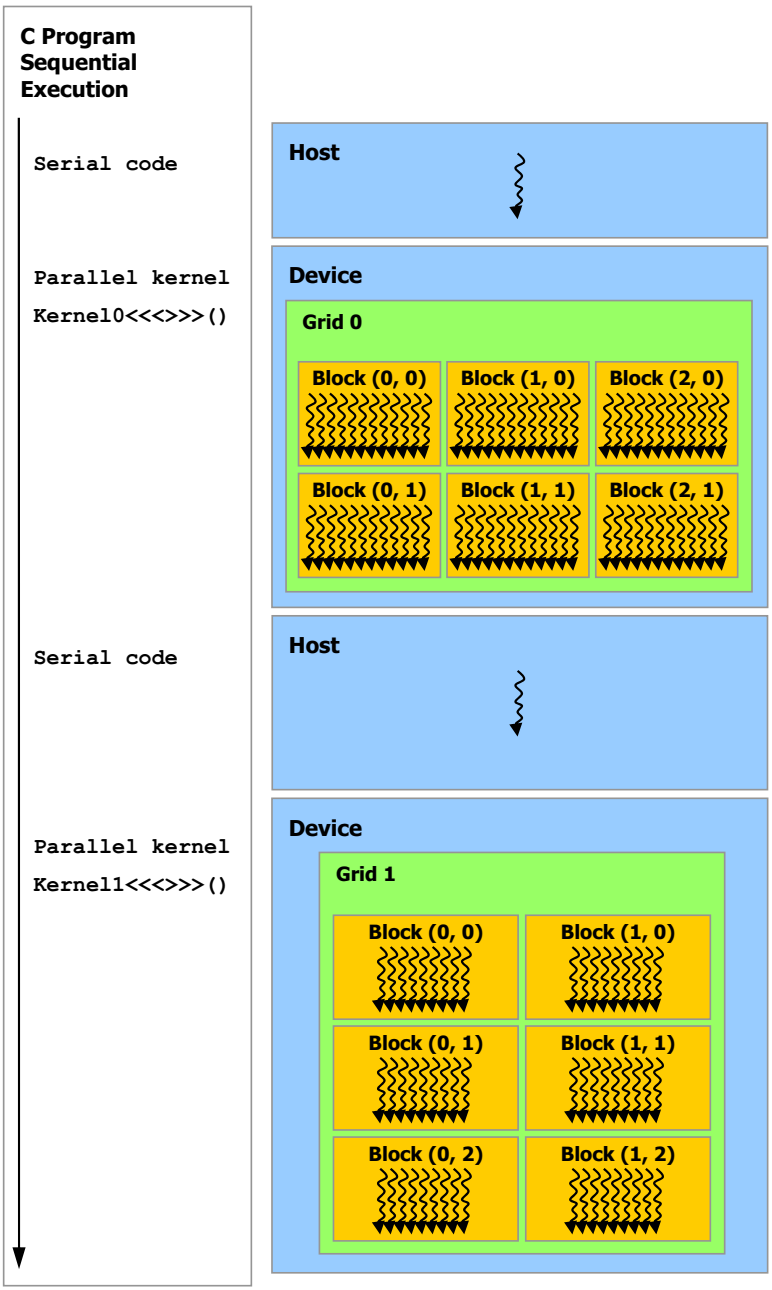

Figure 3. CUDA Program Structure[2]). Serial code executes on the host (CPU) while parallel kernels execute on the device (GPU).

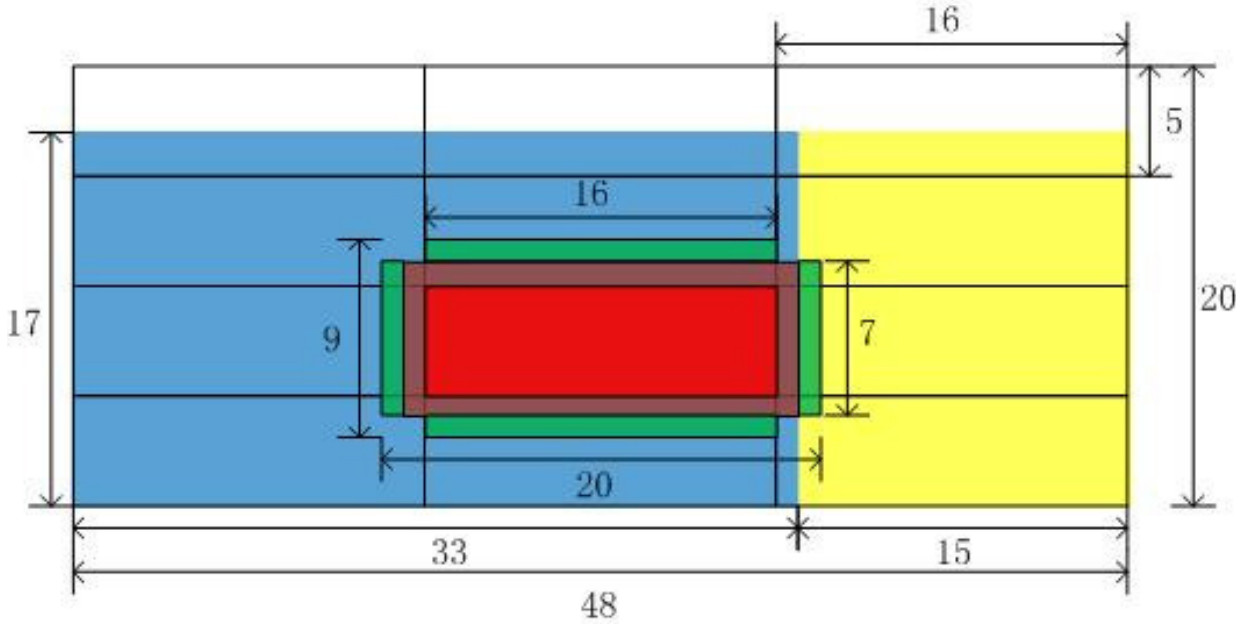

Figure 4. Domain Layout for GPU solution of Euler Equations. Blue: domain; Yellow: row padding; Red: updated region; Brown: compute overlap; Green: smoothing overlap.

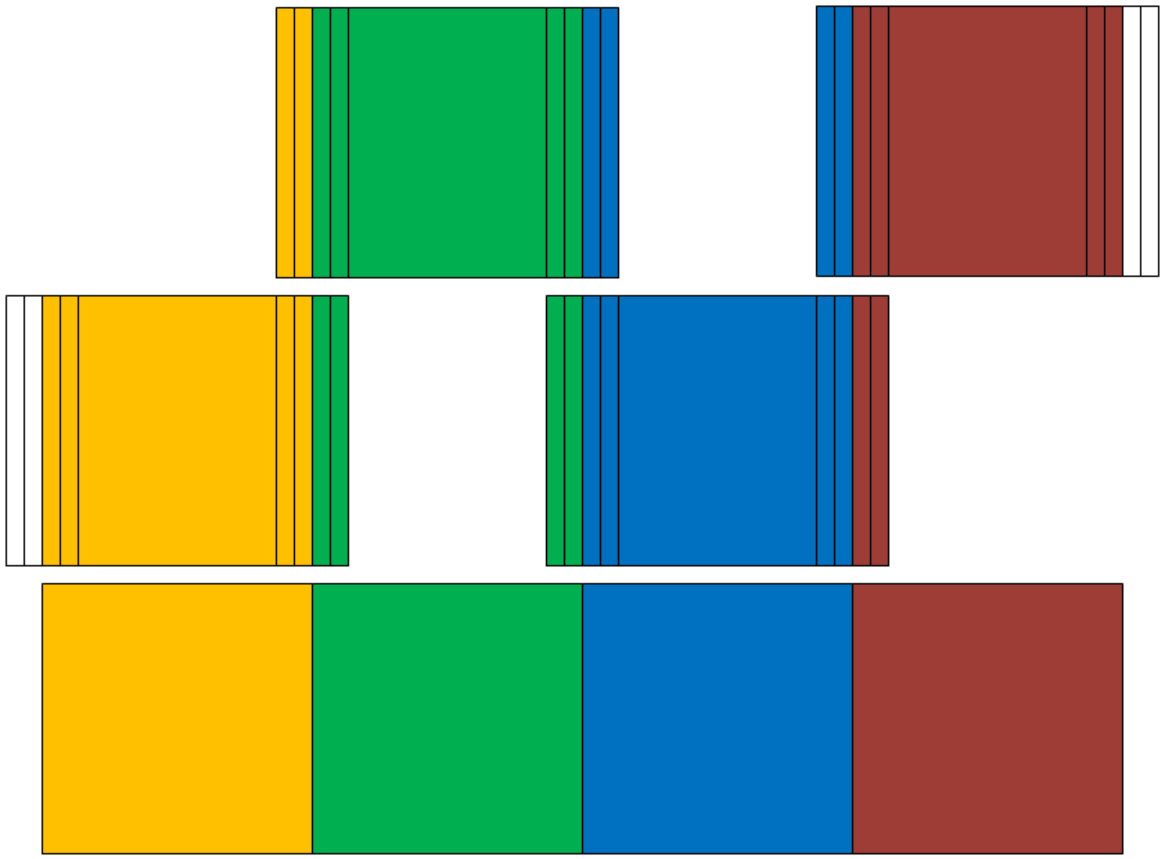

Figure 5. A simple 1-D domain decomposition across multiple GPUs in the cluster. Each subdomain communicates two lines of data with neighbors for smoothing purposes.

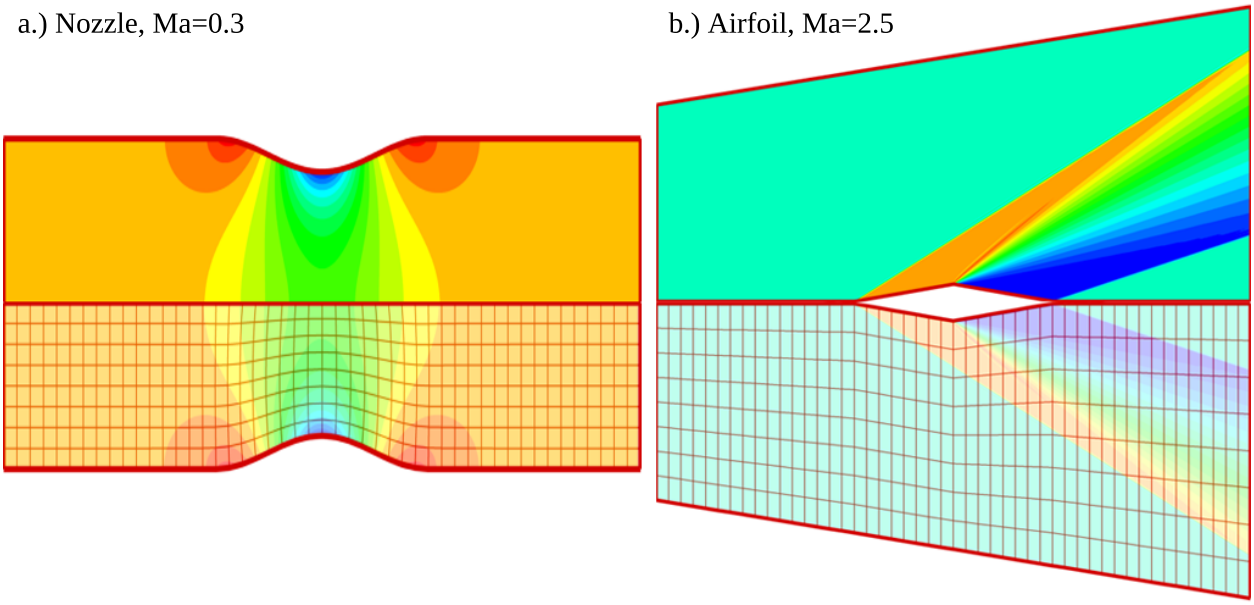

Figure 6. Results of GPU Euler solver showing pressure contours and grid geometry in a nozzle and over a supersonic diamond airfoil. The Airfoil case demonstrates the shock capturing capability of the algorithm.

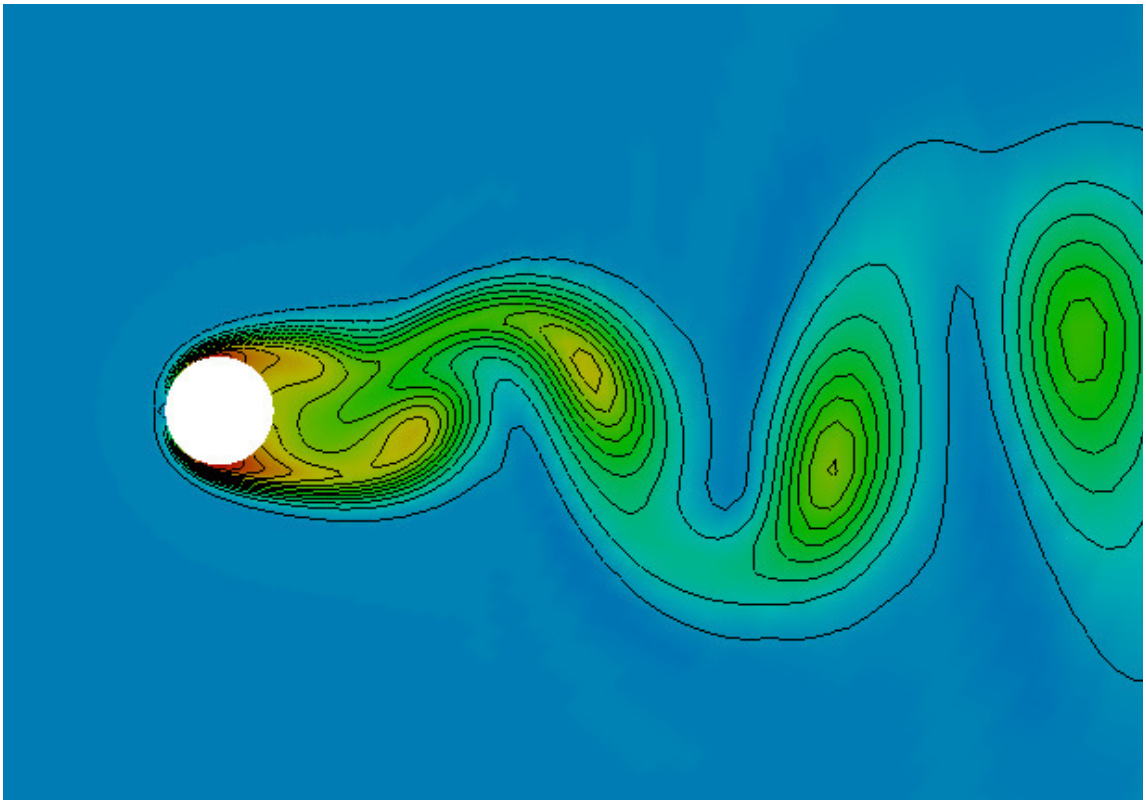

Figure 7. Results of MBFLO showing entropy contours in an unsteady compressible laminar flow over a cylinder at Mach Number 0.10 and Reynolds number 140.

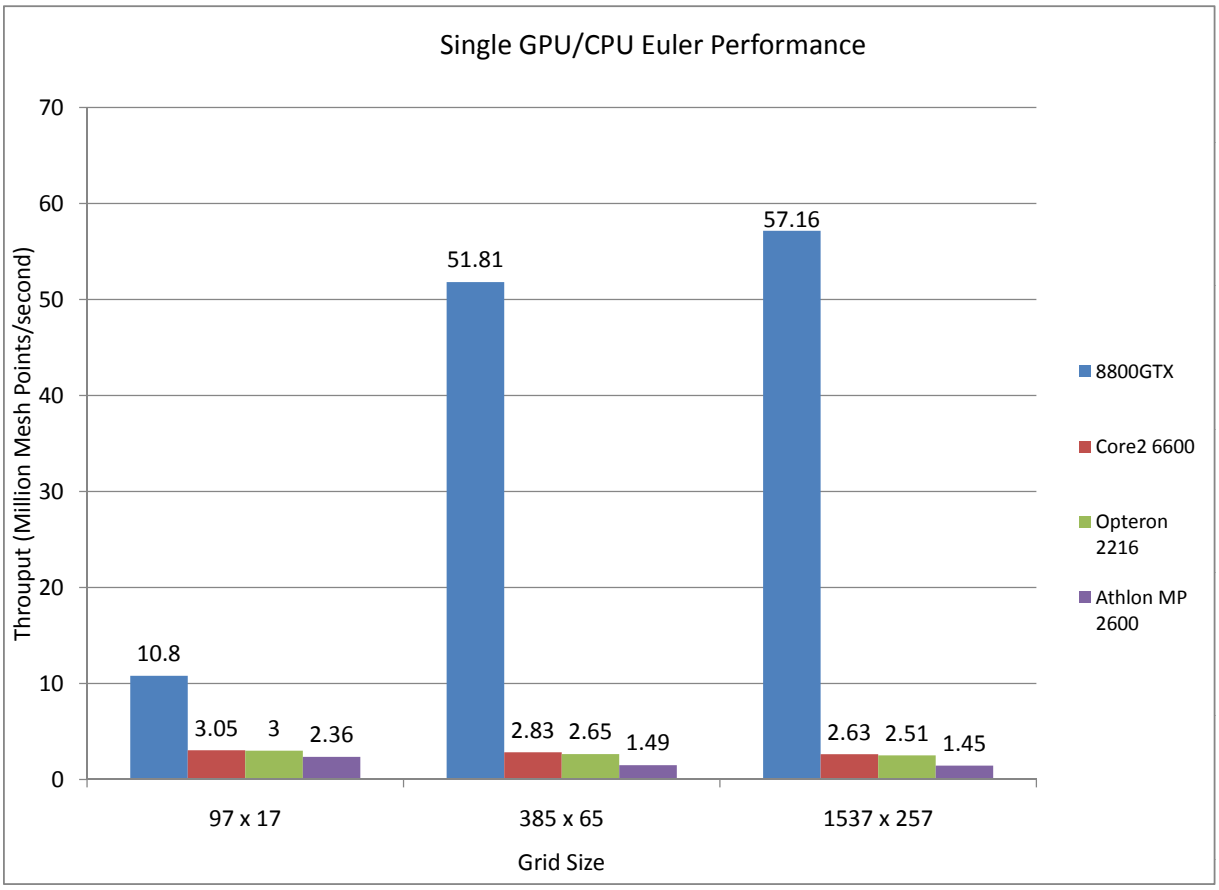

Figure 8. Serial Euler solver performance for nozzle test case of varying grid size.

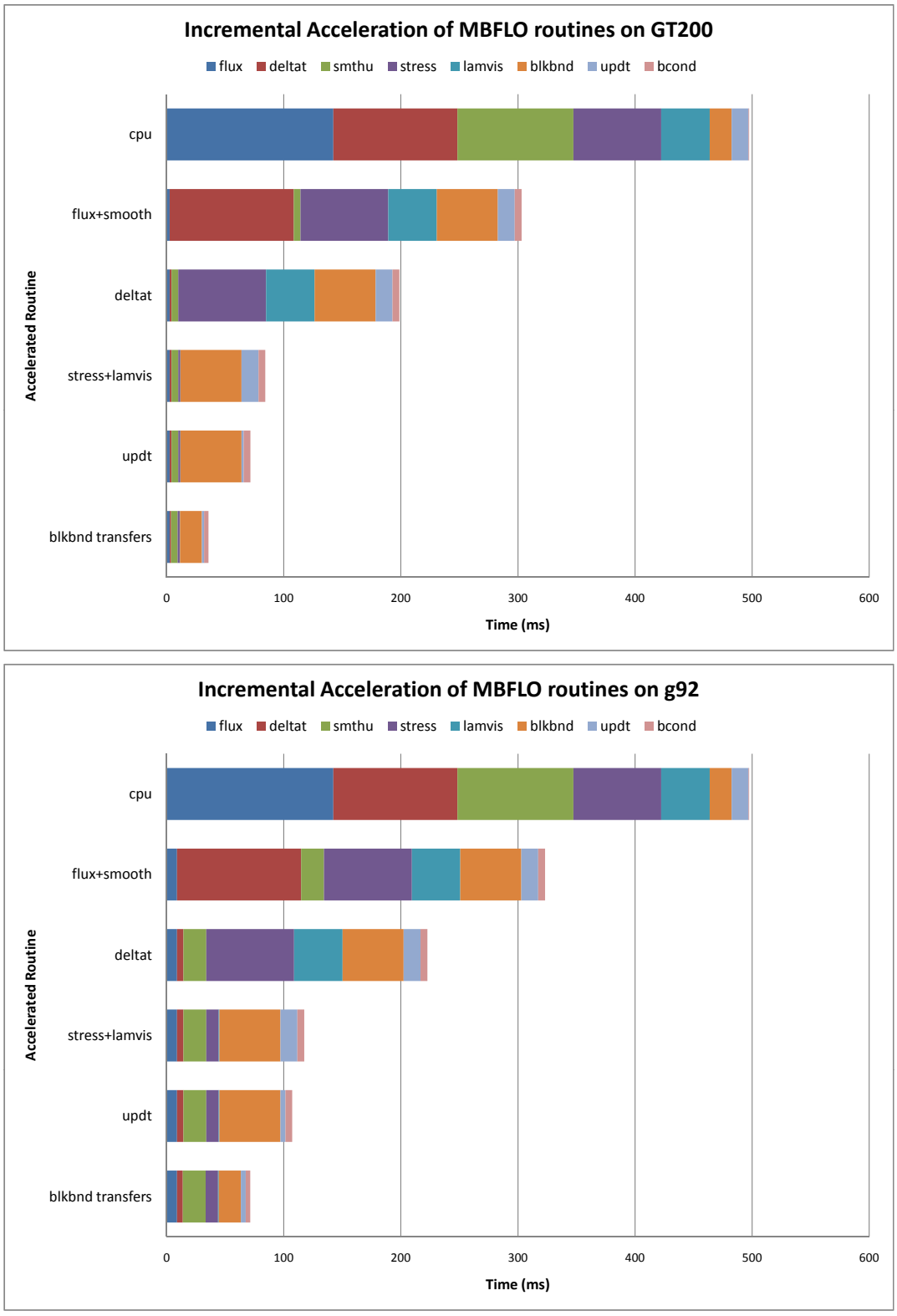

Figure 9. MBFLO acceleration of the main routines is done incrementally by adding GPU support to more of the subroutines and optimizing data movement between CPU and GPU. The above timing results are obtained on a steady laminar flow cylinder test case with a mesh size of 1025 $\times$ 257.

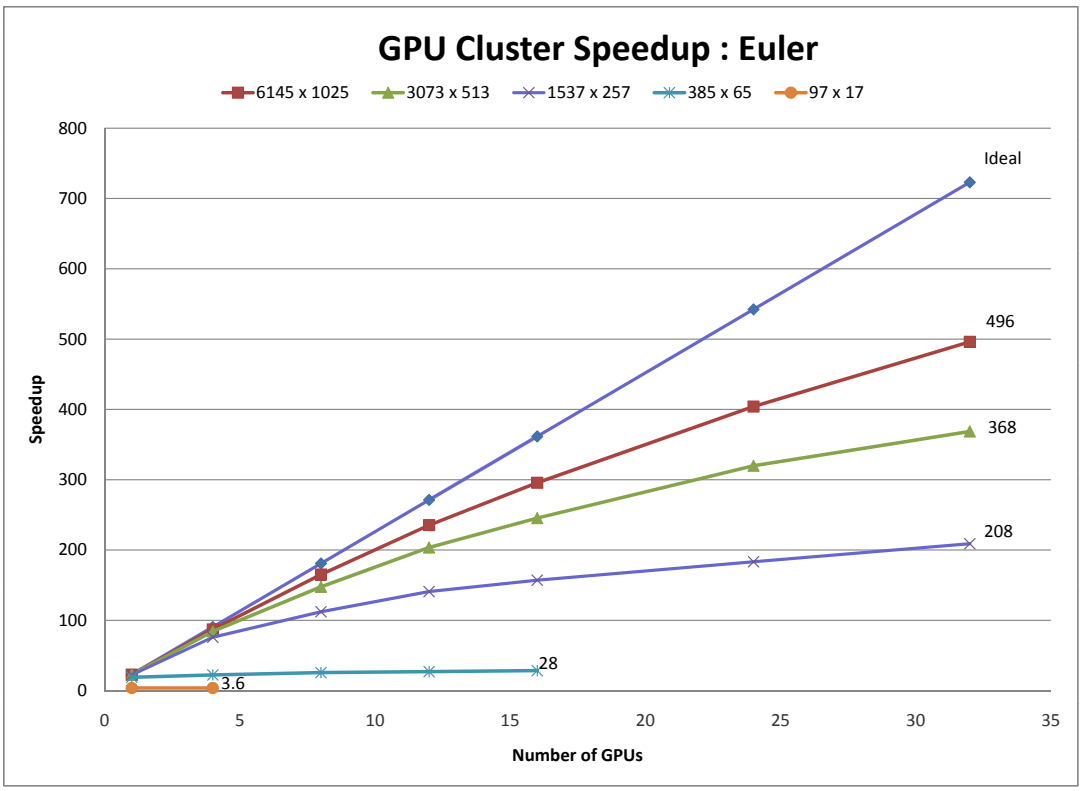

Figure 10. Parallel Euler solver performance for varying mesh size Nozzle test cases relative to a single core of a Q9300 CPU. Multiple GPU performance benefits are not realized until the domain reaches 1537×257 grid points. As the domain size is increased the code scales better, reaching a maximum speedup of 496x with 32 GPUs on the 6145×1025 domain.

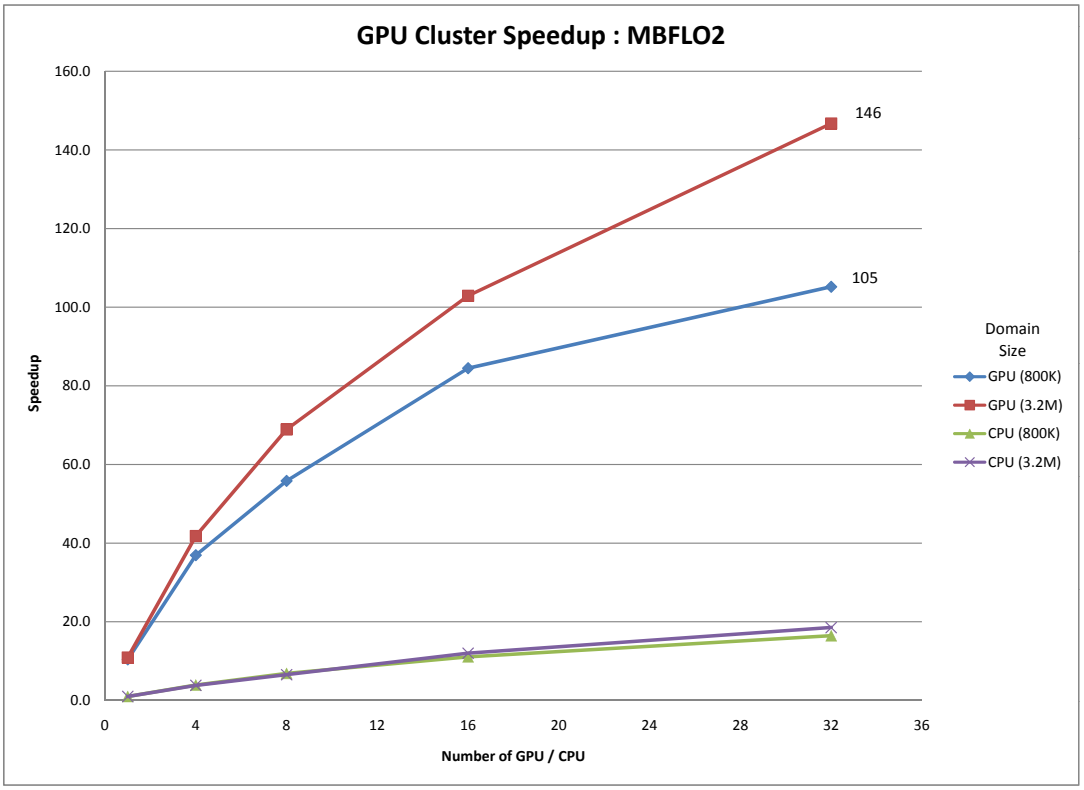

Figure 11. Parallel MBFLO solver performance for varying mesh size unsteady laminar cylinder test cases. 32 GPUs perform up to 146x faster than a single CPU and roughly 8x faster than 32 CPUs.