NASA/TM–2012-217770

# Production Level CFD Code Acceleration for Hybrid Many-Core Architectures

*Austen C. Duffy*
*National Institute of Aerospace, Hampton, Virginia*

*Dana P. Hammond and Eric J. Nielsen*
*Langley Research Center, Hampton, Virginia*

October 2012

## NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 443-757-5803

- Phone the NASA STI Information Desk at 443-757-5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/TM–2012-217770

# Production Level CFD Code Acceleration for Hybrid Many-Core Architectures

*Austen C. Duffy*
*National Institute of Aerospace, Hampton, Virginia*

*Dana P. Hammond and Eric J. Nielsen*
*Langley Research Center, Hampton, Virginia*

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

October 2012

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

## Abstract

In this work, a novel graphics processing unit (GPU) distributed sharing model for hybrid many-core architectures is introduced and employed in the acceleration of a production-level computational fluid dynamics (CFD) code. The latest generation graphics hardware allows multiple processor cores to simultaneously share a single GPU through concurrent kernel execution. This feature has allowed the NASA FUN3D code to be accelerated in parallel with up to four processor cores sharing a single GPU. For codes to scale and fully use resources on these and the next generation machines, codes will need to employ some type of GPU sharing model–as presented in this work. Findings include the effects of GPU sharing on overall performance. A discussion of the inherent challenges that parallel unstructured CFD codes face in accelerator-based computing environments is included, with considerations for future generation architectures. This work was completed by the author in August 2010, and reflects the analysis and results of the time.


# 1 Introduction

High performance computing systems are undergoing a rapid shift towards the coupling of add-on hardware, such as graphics processing units, Cell processors and field programmable gate arrays (FPGAs), with traditional multicore CPU. Hybrid clusters possess unprecedented price-to-performance ratios and high-energy efficiency, having placed in the top supercomputer rankings [1], and dominated the Green500 list [2]. In the US, GPU clusters are already used for research in large scale hybrid computing, including the Lincoln Tesla cluster at the National Center for Supercomputing Applications [3]. Plans to build many more of these machines in the very near future are already underway, and while new codes can be developed with this in mind, an emerging challenge in this modern heterogeneous computing landscape is updating old software to make best use of newly available hardware. While a code often achieves maximum benefit when designed from the start with GPUs in mind, this is not a viable option for many widely used legacy codes whose development has spanned multiple decades. In these scenarios, the use of accelerator models in which tasks with large amounts of data parallelism are ported to GPU code may provide the best solution. While some GPU codes can demonstrate multiple orders of magnitude speedup, additional constraints will likely prevent existing high level computational fluid dynamics (CFD) codes from realizing these gains, particularly when they have already been highly optimized for large scale parallel computation, or when the underlying spatial discretization is unstructured. Additionally, architectures with many CPU cores typically need to share GPU resources leading to further limits on the increased performance of a code

utilizing an entire cluster. The goal of the current work is to examine the appropriateness of hybrid many-core computers for use in computation of a large scale unstructured CFD legacy code, and to identify bottlenecks or limitations that need to be addressed on the path towards exascale computing. In this paper we will examine the acceleration of NASA's FUN3D code [4] with a novel GPU distributed sharing model, and discuss the current challenges that face production level unstructured CFD codes in accelerator based computing environments.

### 1.1 Related Work

There have been a number of papers demonstrating the impressive performance gains GPU computing can provide to applications across a wide range of science, engineering and finance disciplines. For brevity, we will mention here just those most closely related to the FUN3D unstructured Navier-Stokes code. Some relevant works that have ported CFD codes to GPUs can be found in [5] and [6], who achieved up to 40X speedups for the compressible Euler equations, the latter on an unstructured grid. While both demonstrated significantly faster GPU code times, we note that neither solved the full compressible Navier-Stokes equations, and both used explicit Runge-Kutta solvers more suitable for acceleration than the implicit algorithm in FUN3D. In addition, neither of these works considered scaling to multiple processors, giving single-core, single-GPU results against individual CPUs. The report by Jespersen [7] describes the porting of NASA's OVERFLOW code to the GPU; but again only for a single core setup and also in a structured grid environment, which has more favorable memory access patterns. Jespersen [7] replaced the 64-bit implicit SSOR solver with a 32-bit GPU Jacobi solver, resulting in a 2.5–3X speedup for the solver, and an overall code wall clock time reduction of 40%, representing more realistic results for a high end CFD code. The work by Cohen and Molemaker [8] is an excellent resource for anyone considering the development or porting of CFD code for GPU computation, and includes double precision considerations, an important aspect of CFD in general. We note that accelerating solvers on multi-GPU desktops [9] is an interesting topic with enormous benefits, as researchers have supercomputing power on hand without the need to share resources. We, however, are more concerned with the need to accelerate highly scalable CFD codes on large clusters with many cores. Reviewing the literature one can see that, until recently, the vast majority of the research in this field has focused solely on utilizing a single CPU core with one or more GPUs as the primary source of computational power. Conversely, only a minor amount of research has considered larger scale computations on many cores with supporting GPUs, or even sharing a GPU among several CPU cores. The latter is due to the fact that until the recent release of NVIDIA's latest architecture Fermi™cards, concurrent kernel execution was not

possible, and hence cores had to wait serially for access to a shared GPU. Gödekke and Strzodka [10–13], among others, have made a substantial contribution to large scale computing on graphics clusters over the last several years, with particular focus to the areas of multigrid and finite element methods. We note some other recent works that have considered using CFD codes in a GPU cluster environment including [14] where a 16 GPU cluster was used to achieve 88X speedup over a single core on a block structured MBFLO solver, and [15] who achieved 130X speedup over 8 CPU cores for incompressible flow with 128 GPUs on the aforementioned Lincoln Tesla cluster. At this time we know of no published works employing any type of GPU sharing model.

### 1.2 Challenges

When attempting to develop an accelerated version of a large parallel CFD code, several challenges present themselves. To start with, these codes have already been highly optimized for coarse-grain parallelism usually involving multi-level cached based architectures. One must look beyond coarse-grain parallelism (e.g., dividing a grid into blocks and distributing them over multiple CPU cores) and seek fine-grained data parallel tasks that can be ported to the GPU. If these opportunities exist for a large scale code, it has the potential to benefit from an added level of parallelism. Parallel codes also employ MPI communications, which when used in an accelerator environment will implicitly invoke expensive GPU-CPU data transfers. Scalability is also a big issue for all production level CFD codes. With the rising number of CPU cores per processor coupled with the rising cost and power consumption of an individual GPU, the probability that future large scale hybrid architectures will be able to maintain a 1:1 or better ratio of GPUs to CPU cores is not likely. The new availability of concurrent kernel execution allows multiple cores to share a single GPU. This feature should allow hybrid codes to scale on systems with a GPU to CPU ratio below 1. This feature does come at a price though, as increasing the number of CPU threads which simultaneously share a single GPU puts added demand on the available memory and could put additional strains on other resources for large kernels. In addition to these challenges, FUN3D's unstructured-grid solver is hindered by the unavoidable constraint of out-of-order memory access patterns, which can cause very poor performance on a GPU. In the remainder of this paper we will give a description of the FUN3D solver along with the steps taken to port it for GPU computation, followed by a results and discussion.

## 2 FUN3D Code

References [16] and [4] describe the software used in the current work. FUN3D consists of several hundred Fortran95 modules containing ap-

proximately 850,000 lines of source code. The software can be used to perform steady or time-dependent aerodynamic simulations across the speed range. Additionally, an extensive list of options and solution algorithms is available for spatial and temporal discretizations on general static or dynamic mixed-element unstructured meshes which may or may not contain overset grid topologies. Options for performing mathematically-rigorous mesh adaptation [17] and formal design optimization [18] using a discrete adjoint formulation are also included. The package has been distributed to hundreds of organizations and is routinely used by groups in academia, industry, and various government agencies.

### 2.1 Mathematical Formulation

The current work focuses on the solution of the compressible Reynolds-averaged Navier-Stokes equations. For the current study, the spatial discretization uses a finite-volume approach in which the dependent variables are stored at the vertices of tetrahedral meshes. Inviscid fluxes at control volume interfaces are computed using the upwind scheme of Roe [19], and viscous fluxes are formed using an approach equivalent to a finite-element Galerkin procedure [16]. For turbulent flows, the eddy viscosity is modeled using the one-equation approach of Spalart and Allmaras [20]. An approximate solution of the linear system of equations formed within each time step is obtained through several iterations of a multicolor point-iterative scheme. The turbulence model is integrated all the way to the wall without the use of wall functions and is solved separately from the mean flow equations at each time step with a time integration and linear system solution scheme identical to that employed for the mean flow equations. In the current implementation, the grid nodes are numbered using a reverse Cuthill-McKee technique [21] to improve cache performance during flux and jacobian gather/scatter operations. However, the multicolor point-iterative scheme forbids physically adjacent unknowns from residing within the same color group. Therefore, the relaxation scheme is inherently cache unfriendly in its basic form. Since the majority of the floating point operations required to advance the solution take place in the relaxation phase, a mapping is introduced which orders the data in the coefficient matrix by its assigned color as the individual contributions to the jacobian elements are determined. In this manner, memory is accessed in ideal sequential order during the relaxation as would be achieved in a more simplistic scheme such as point-Jacobi, while the substantial algorithmic benefits (improved stability and convergence) of the multicolor scheme are retained.

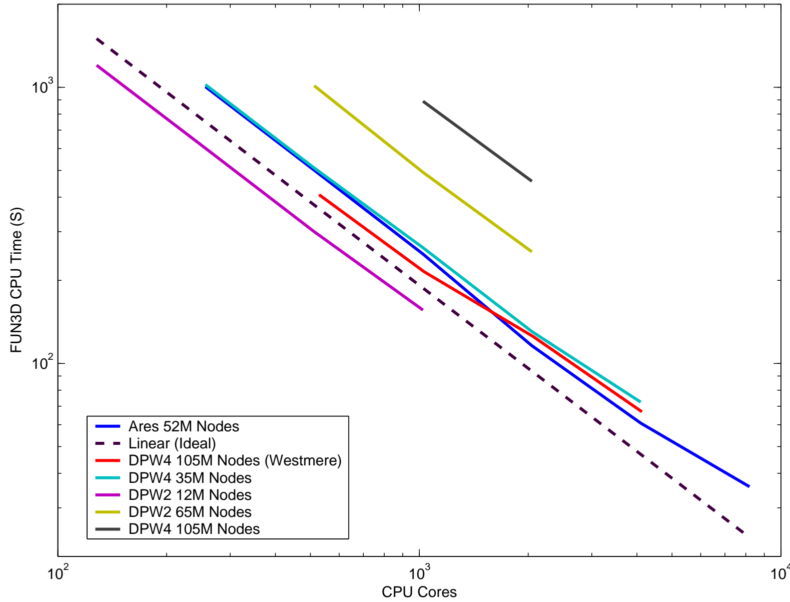


Figure 1. Parallel scaling results for FUN3D. Problems are given by node size, with M equating to millions of nodes. There are approximately 6 times as many tetrahedral cells as there are nodes for a given problem size. Ares is a rocket geometry and DPW grids represent aircraft configurations from AIAA Drag Prediction Workshops. The linear dashed line represents ideal scaling.

## 2.2 Parallelization and Scaling

Parallel scalability to thousands of processors is achieved through domain decomposition and message passing communication. Pre- and post-processing operations are also performed in parallel using distributed memory, avoiding the need for a single image of the mesh or solution at any time and ultimately yielding a highly efficient end-to-end simulation paradigm. Typical scaling performance for the solver on a range of mesh sizes is shown in Figure 1, with meshes in the millions of nodes. A mesh typically contains six times as many tetrahedral cells than the number of nodes, and so the 105 million node mesh equates to 630 million tetrahedral cells. The majority of these results have been generated on a SGI®Altix®ICE platform consisting of dual-socket, quad-core Intel®Xeon®"Harpertown" processors. The data marked "Westmere" has been generated using dual- socket, hex-core "Westmere" processors. The implementation scales well across the range of processing cores shown.

One factor in maintaining scalable performance within FUN3D is the load balancing of the computation and communication costs over the processors, which requires efficient partitioning of the underlying unstruc-

tured grid. The mesh partitioning software Metis and ParMetis [22, 23] is used by many CFD codes. Metis' partitioning objectives attempt to evenly distribute the number of nodes (work) to each processor, while minimizing the number of adjacent elements assigned to different processors (communication). In graph theory, these objectives translate to minimizing the edge-cuts and minimizing the total communication volume.

## 3 GPU Acceleration

To improve the performance of the FUN3D code, a minimally invasive accelerator model has been implemented in which code portions are ported to the GPU. This allows for an increase in speed without altering the proven methods of the underlying solvers. The FUN3D code portion targeted for acceleration was the point implicit subroutine used for groups of 5x5 matrix equations which represent the linearized form of the discrete mean flow equations. These correspond to the density, velocity and pressure variables $\rho$, $u$, $v$, $w$, and $p$. This routine can account for as much as 70% of the program's total CPU time (such as when solving the Euler equations), but is typically closer to 30% (RANS), depending on the required number of sweeps by the colored Gauss-Seidel (GS) solver for the particular problem. Cache size can play a significant role on this routine and on unstructured codes in general. For this reason along with the need for concurrent kernel execution capabilities, only the latest generation NVIDIA Fermi(TM) architecture hardware is targeted, since these possess true L1 and L2 caches which were previously unavailable. On the CPU side, each core performs the colored GS solve sequentially, so a natural mapping to the GPU is created with single threads computing the solutions to the individual 5x5 block solves of a color in parallel. A CUDA C subroutine call has been inserted into the original Fortran `point_solve_5` (PS5) subroutine to carry out the code porting. The computation of the right hand side (RHS) solve between colors has effectively been moved to the GPU when sufficient available memory and double precision capabilities are present. A general outline of PS5 is given in Algorithm 1, noting single precision (SP), double precision (DP) and mixed precision (MP) segments. For the test problems, 10,000s to 100,000s of threads are launched, each computing an entire 5x5 block solve corresponding to an individual equation for the particular color. CUDA C was chosen because it allows for the most explicit user control over the previously mentioned Fermi architecture GPUs that we are employing as accelerators. CUDA C also provides portability as it is widely used and the compiler is freely available. Given the code, one could (with a little effort) extend this capability to other accelerators with OpenCL, as the language extensions are similar. In general, any accelerator used must have capabilities for both concurrent kernel exe-

cution and double precision arithmetic, though there are exceptions that will be discussed further.

**Algorithm 1** PS5 Subroutine Outline

```
for color = 1 to max_colors do
  b(:) = residual(:)
  // Begin RHS solve
  for n = start to end do
    Step 1. Compute off diagonal contributions
    for j = istart to iend do
      b_i = b_i − Σ_{i≠j} A_{i,j} ∗ x_{icol} (SP)
    end for
    Step 2. Compute 5x5 forward solve (DP)
    Step 3. Compute 5x5 back solve (DP)
    Step 4. Compute sum contribution, update RHS (MP)
    Step 5. Accumulate Sum Contributions (GPU Only)
  end for
  // End color, do MPI communication
  Call MPI transfer
end for
```

### 3.1 GPU Distributed Sharing Model

The GPU code has been developed with four different CUDA subroutines; the first dedicated for a single core-single GPU scenario, and the remaining three for multi-core scenarios that require MPI communication transfers at the end of each color sweep. The MPI versions are based on what we call a GPU distributed sharing model. We will define this as an accelerator model where work is efficiently distributed across multiple processors that share a single GPU. This should be done in such a manner that, if necessary, the CPU cores perform some amount of the computational work that would otherwise be done by the GPU to ensure that kernel execution remains optimal. This is an important concept since large kernels can place large demands on available GPU resources when called simultaneously, particularly when they use many registers and large chunks of shared and constant memory. In fact, if too many resources are required, kernels from multiple threads could be scheduled to execute sequentially, causing a substantial or complete loss of performance gains. Sharing kernel work with multiple CPU threads should also reduce core idle time, leading to more resource efficient execution. One constraint we have found inherent to this model is a reduction in available GPU memory. As data is distributed to multiple CPU cores, there are more individual structures that are padded when put into GPU memory, leading to a reduction in the available global memory for each additional thread. In all of our test cases on a GTX 480 card, each

additional thread sharing the GPU reduces the available global memory predictably within the 63–68 MB range, independent of problem size. This yields an average loss of approximately 66.7 MB of available memory per shared CPU thread, for exactly the same amount of data being stored.

We consider GPU sharing to be an important feature of new and future hybrid CFD codes. We recognize that on a large shared hybrid cluster one could simply use one core per GPU, allowing the remaining cores of a node to be allocated to other tasks. However, with a scalable CFD code it is optimal to utilize as many cores as possible, since this should lead to the best overall performance. Consider running a CFD code that easily scales to thousands of cores on a 256 GPU cluster. Clearly, one would produce better times with 1,024 cores and 256 GPUs than with only 256 cores and 256 GPUs, even if the speedup over the CPU cores alone was not as dramatic.

### 3.2 CUDA Subroutines

We will now present the four new CUDA subroutines introduced into the FUN3D code, and note that only solution data transfers to and from the GPU are required within the subroutine itself. All other necessary data is transferred externally in optimal locations. Also note that these subroutines only replace a single sweep of the colored GS solver, as MPI communications are typically necessary after every sweep.

- `gs_gpu` : The first subroutine is for a straightforward 1 core to 1 GPU setup where the entire GS solve is done in CUDA. The kernel for this scenario computes the entire RHS solve, and is called repeatedly through the color sweeps without any data transfers. In this case, a single GPU thread maps to a single control volume, computing its own 5x5 block solve (steps one through four in Algorithm 1). After a thread synchronization, the kernel keeps five threads to accumulate the sum contributions for the five variables (step 5 of Algorithm 1) before moving on to the next color. This scenario does not require MPI communication and hence provides the biggest performance gains, but is limited to a single processor. We will refer to this routine as `gs_gpu`, which does not require concurrent kernel execution. The kernel for this routine allocates 21 registers per thread, and 136 bytes of constant memory per thread block at compile time, as found by the `--ptxas-options=-v` compiler option. It requires one CPU to GPU solution transfer at the beginning of the call, and one GPU to CPU transfer at the end, but none between colors. We note that our unstructured code is unable to effectively use the 48 KB shared memory space, and opt instead to swap this larger section with the 16 KB L1 cache space using the `cudaFuncSetCacheConfig` function. Using the larger L1

cache space provides a noticeable 6% performance increase over the smaller version for this routine.

- `gs_mpi0` : This case is identical to that above in terms of the GPU kernel, however, it assumes that an MPI communication is required at the end of each color. Here, the CUDA routine must copy the solution back from the GPU and return it to the Fortran subroutine to perform the transfer, providing additional overhead. We will refer to this routine as `gs_mpi0`, which uses the same kernel as `gs_gpu` and hence maintains identical GPU resource allocation properties. This routine, along with the other MPI CUDA routines, require two additional data transfers (one CPU to GPU and one GPU to CPU) for each color.

- `gs_mpi1` : This subroutine is the first to consider the effects of GPU sharing, by attempting to reduce the kernel size and distribute a small portion of the workload to the CPU. When developing kernels, one must try to reduce CPU-GPU memory copies at all costs, even if that means performing less than ideal tasks on the GPU side. The kernel of the first two subroutines performs a final accumulation of sum contributions with a mere 5 threads; and so this portion of code does not benefit from mass thread parallelism, but does reduce transfer overhead by keeping sum contributions on the GPU. As more threads share the GPU, this potential overhead can be reduced since multiple CPU cores can copy their respective sum contributions back in parallel. Once there, the more powerful CPU cores should be able to perform the accumulations faster. This concept is implemented in subroutine `gs_mpi1`. Moving this small amount of work to the CPU reduces the constant memory by a mere 8 bytes and does not reduce the kernel register count.

- `gs_mpi2` : The final subroutine was designed to execute with many threads sharing a single GPU. It shares a substantial amount of computation with the CPU, and also eliminates all double precision calculations from the GPU side. This is accomplished by cutting the kernel short and computing the double precision forward and backward solves along with the sum and solution updates on the CPU, only computing step 1 of Algorithm 1 on the GPU. In the process, the need to transfer the diagonal of a matrix is eliminated, leading to a reduction in data transfer costs and more importantly, freeing up valuable global memory space. The kernel for this subroutine reduces the register count by 1, while maintaining the same constant memory requirements as the kernel of `gs_mpi1`.

| Cores | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 339,206 | | | | | | | |
| 2 | 168,161 | 171,045 | | | | | | |
| 4 | 85,339 | 84,626 | 84,861 | 84,380 | | | | |
| 8 | 42,824 | 40,608 | 42,828 | 42,622 | 42,555 | 42,525 | 42,726 | 42,518 |

Table 1. The grid partitioning shows that the grid-node distributions are well balanced as the number of processors vary.

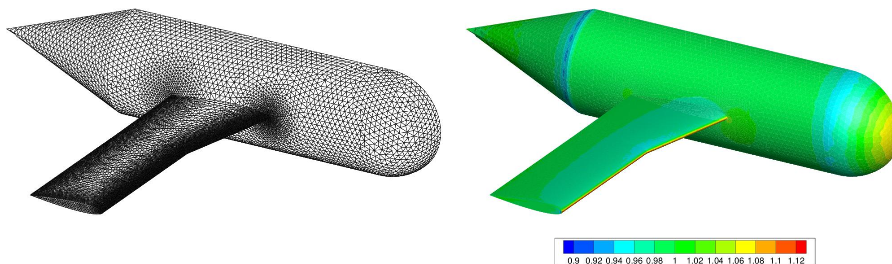


Figure 2. Test problem 1 grid (left) and GPU computed pressure solution for $p/p_\infty$ (right). Generic wing-body geometry, contains 339,206 nodes and 1,995,247 tetrahedral cells. Pressure solution obtained at convergence of the FUN3D solver (132 iterations). Inviscid flow, Mach Number = 0.3, Angle of Attack = 2.0 degrees.

# 4 Test Problems

The GPU accelerated FUN3D code has been tested with a number of problem sizes ranging from approximately 16,000 to 1.1 million grid nodes (96,000–6.6 million elements). This section describes two particular test problems in details; the first case is inviscid, the second is viscous.

## 4.1 Test Problem 1

The first test problem is a generic wing-body geometry, which is a pure tetrahedral grid with 37,834 triangular boundary faces, 339,206 Nodes and 1,995,247 Cells. The node partitioning is given in Table 1, and the surface grid is shown in Figure 2 along with the GPU-computed pressure solution for $p/p_\infty$ at convergence. This setup calls 20 PS5 sweeps per iteration, accounting for approximately 70% of the FUN3D total run time. This problem is inviscid, has a Mach number of 0.3 and an angle of attack of 2.0 degrees.

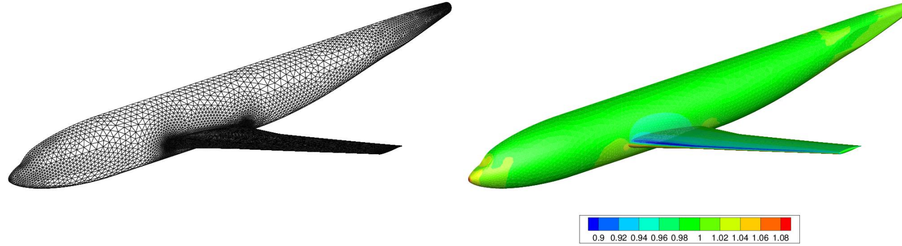


Figure 3. Test problem 2 grid (left) and GPU computed pressure solution for $p/p_\infty$ (right). DLR-F6 wing-body configuration, contains approximately 650,000 nodes and 3.9 million tetrahedral cells. Turbulent flow, Mach Number = 0.76, Angle of Attack = 0.0 degrees, Reynolds Number $= 1,000,000$.

### 4.2 Test Problem 2

The second problem is a DLR-F6 wing-body configuration used in the second international AIAA Drag Prediction Workshop (DPW-II) [24]. The versions used here include the coarse grid containing 1,121,301 nodes and 6,558,758 tetrahedral cells, and also a reduced grid with approximately 650,000 nodes and 3.9 million tetrahedral cells. In this case, PS5 corresponds to approximately 30% of the total FUN3D wall clock time. The smaller grid along with the corresponding GPU computed pressure solution for $p/p_\infty$ are given in Figure 3, corresponding to a turbulent solution with a Mach number of 0.76, an angle of attack of 0.0 degrees and a Reynolds number of 1 million.

## 5 Results and Discussion

Two machines were used for timing studies: a dual socket Dell workstation with Intel Xeon 5080 CPUs (2 cores @ 3.73 GHz, 2 MB L2) and a single NVIDIA GTX 480 GPU. The second machine is a small Beowulf GPU cluster with a head node and two Fermi equipped compute nodes connected by a gigabit switch. Each compute node possesses an AMD Athlon II X4 620 processor (4 cores @ 2.6 GHz, 2 MB L2) and a single NVIDIA GTX 470 card.

### 5.1 Single Core Performance

Performance results for a single sweep of the PS5 solver on the GTX 480 workstation for a range of grid node sizes are given in Figure 4. The best performing subroutine, `gs_gpu`, achieves a speedup of approximately 5.5X regardless of problem size. Performance for the `gs_mpi0`

routine which utilizes the same CUDA kernel is cut in half as a result of CPU-GPU data transfers necessary for MPI communication between colors. We see that version `gs_mpi2` comes closest to the performance of `gs_gpu`, and believe that this is due to the removal of double precision calculations, as the GeForce series cards have been purposely de-tuned for these. We speculate that `gs_mpi1` would achieve the best MPI performance in the presence of higher end Tesla series cards with full double precision capabilities. Single core speedups on the Beowulf cluster are not as good, with a best case of 4.75X. This is due to the fact that the CPU times are slightly better on a single AMD core, and that the GTX 470 is slightly less capable than the 480 model. For the single core scenario on both machines, the overall FUN3D code is accelerated by a factor of 2X or more for test problem 1, which spends about 70% of its total CPU time in the PS5 subroutine. We note that our CUDA subroutines execute the fastest with only 8–16 threads per block, well below the NVIDIA minimum recommendation of 64. This is likely due to the additional cache resources available to each thread. While we are primarily concerned with large scale parallel applications, these single core results provide valuable insight. We have learned here that even in the absence of MPI transfers, speedups are limited and well below the bar set by so many other GPU accelerated applications. We do note, however, that larger cache sizes could help narrow the gap between this unstructured code and those that benefit from ordered memory access patterns. We also find that in a single core setting our GPU accelerated implicit solver has exceeded the 2.5–3X speedup achieved by that of the OVERFLOW code [7], perhaps a more fitting standard for comparison.

### 5.2 Multiple Core Performance

The Beowulf GPU cluster allows us to study scenarios up to 8 cores, 2 GPUs. Figures 5 and 6 show strong and weak scaling results. Strong scaling within a single node shows the limitations of a GPU distributed sharing model, namely, that if developed properly, the GPU code should execute at approximately the same speed regardless of the number of cores per GPU. Due to this factor, local CPU scaling will ultimately provide the limit on the number of cores that can share a single GPU. We see here from the strong scaling figure that a core limit is not reached for this machine, but it would likely be four with more cores present. Weak scaling results are comparable to the original code on two nodes, but testing on a larger cluster is needed to provide better insight. Viewing the weak scaling results, we find that in a grid node for grid node manner the PS5 subroutine runs about 40% faster when the nodes employ GPUs. This may not seem significant considering the high expectations of GPU accelerated codes, but one must consider that the code is unstructured, there are 20 or more CPU-GPU solution transfers per sweep (one to the GPU and one back to the CPU for each GS color), and this is achieved

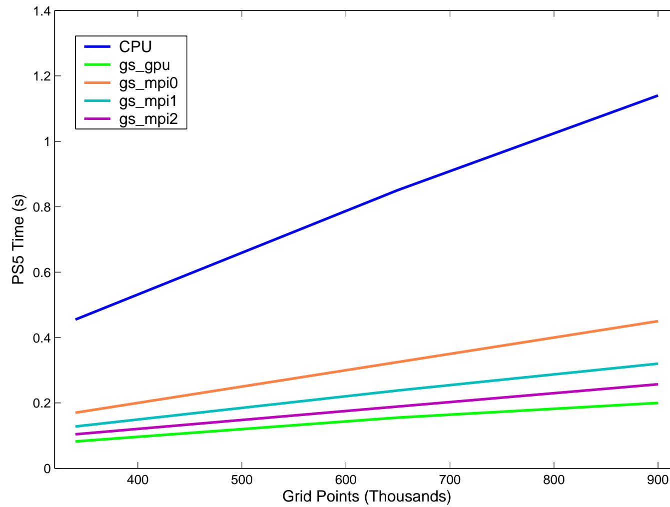


Figure 4. CPU and GPU performances for a single PS5 sweep with varying test case sizes. Tests were run on a single core of an Intel Xeon 5080 CPU mated with a single NVIDIA GTX 480 GPU.

with four processor cores sharing a single low cost (about $350) gaming card.

### 5.3 Discussion

While the results obtained in this work are limited to a 2X overall code speedup, they do indicate future potential for the use of large GPU clusters with production level unstructured CFD codes. Considering that GPU chips are advancing at a faster pace than CPU versions, this code may perform substantially better on the next generation architecture with larger cache sizes and faster memory access speeds. Table 2 provides a look at the rapid advancements occurring in GPU technology, which is beginning to noticeably outpace those of CPUs. Should these trends continue with more available cache and faster memory access speeds, hybrid clusters will become much more amenable to the acceleration of unstructured codes. However, if GPUs are to play an integral role in large scale parallel computations involving high level codes, data transfer technologies will also need to improve to reduce the performance penalty incurred through CPU-GPU data transfers, as these will provide the biggest bottleneck.

## 6 Conclusions

The FUN3D code has been accelerated in a parallel setting with as many as 4 processor cores sharing a single low cost GeForce series GPU by uti-

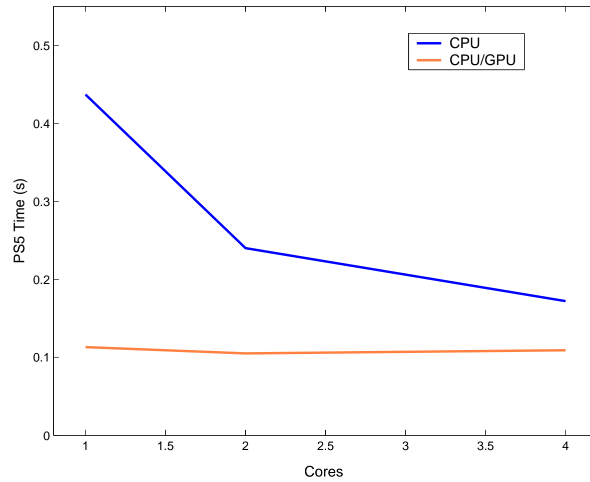


Figure 5. Strong scaling within a single node for a single sweep of the PS5 subroutine using CUDA version gs_mpi2. Tests were run on 2 nodes of a Beowulf GPU cluster, each running an AMD Athlon II X4 quad core processor and NVIDIA GTX 470 GPU. Strong scaling results use a 340,000 grid node test problem on one node.

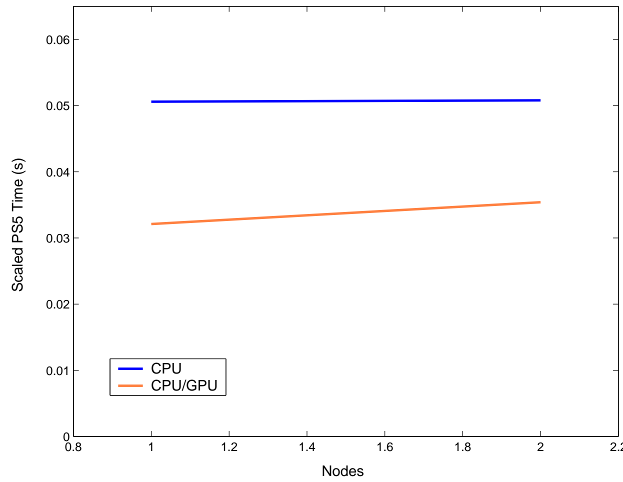


Figure 6. Weak scaling across two compute nodes for a single sweep of the PS5 subroutine using CUDA version gs_mpi2. Tests were run on 2 nodes of a Beowulf GPU cluster, each running an AMD Athlon II X4 quad core processor and NVIDIA GTX 470 GPU. Weak scaling results use a 340,000 grid node problem on one node, and a 650,000 grid node problem distributed across 2 nodes. Weak scaling times are per 100,000 grid nodes.

| Architecture | Year | Cores | L1 Cache | L2 Cache | Memory Access Speed |
|---|---|---|---|---|---|
| G80 | 2006 | 112 | 16 KB [1] | 128 KB [2] | 57.6 GB/s GDDR3 |
| GT200 | 2008 | 240 | 24 KB [1] | 256 KB [2] | 102 GB/s GDDR3 |
| Fermi | 2010 | 448 | 48/16 KB [3] | 768 KB | 144 GB/s GDDR5 |

Table 2. GPU architecture evolution from G80, which approximately coincided with the release of Intel's quad core CPUs, to Fermi which coincided with the release of Intel's six core processors. GPU advancements over the last few years have noticeably outpaced those of CPUs. Representative GPUs are: G80-GeForce 8800 GT, GT200-Tesla C1060, Fermi-Tesla C2050. [1]shared memory, [2] texture memory, [3]Configurable L1/shared memory.

lizing a novel GPU distributed sharing model. The increased double precision capabilities of a new Tesla series card may provide for acceleration with more cores per GPU, but at the time of this writing, we do not possess the hardware. The introduction of true L1 and L2 cache spaces along with concurrent kernel execution capabilities in the Fermi chip has opened the door for viable acceleration of large scale production level unstructured CFD codes. If current hardware trends continue as highlighted in the discussion, much more meaningful performance gains from this and other unstructured hybrid codes may be realized in the next generation. Ultimately, however, when considering the parallel nature of production level CFD codes it seems necessary that new hardware models should be developed to rapidly increase the speed of CPU-GPU data transfers, as these currently provide a serious bottleneck when scaling to multiple cores. Most advanced numerical methods require frequent sharing of information, and so advanced scalable codes can not be expected to run long GPU calculations without frequent external communication.

## References


1. http://www.top500.org, last accessed Aug. 5, 2010.

2. http://www.green500.org, last accessed Aug. 16th, 2010.

3. Kindratenko, V.; Enos, J.; Shi, G.; Showerman, M.; Arnold, G.; Stone, J.; Phillips, J.; and Hwu, W.: GPU Clusters for High-Performance Computing. *IEEE Cluster 2009*, New Orleans, Louisiana, USA, September 2009.

4. http://fun3d.larc.nasa.gov, last accessed May 22, 2012.

5. Elsen, E.; LeGresley, P.; and Darve, E.: Large calculation of the flow over a hypersonic vehicle using a GPU. *J. Comput. Phys.*, vol. 227, 2008, pp. 10148–10161.

6. Corrigan, A.; Camelli, F.; Lohner, R.; and Wallin, J.: Running Unstructured Grid Based CFD Solvers on Modern Graphics Hardware. *19th AIAA CFD Conference*, San Antonio, Texas, USA, June 2009.

7. Jesperson, D. C.: Acceleration of a CFD Code with a GPU. NAS Technical Report NAS-09-003, NASA Ames Research Center, November 2009.

8. Cohen, J. M.; and Molemaker, M. J.: A Fast Double Precision CFD Code Using CUDA. *Proceedings of Parallel CFD 2009*, 2009.

9. Thibault, J. C.; and Senocak, I.: CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. *47th AIAA Aerospace Sciences Meeting*, Orlando, Florida, USA, January 2009.

10. Göddeke, 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 Comput.*, vol. 33, no. 10-11, 2007, pp. 685–699.

11. Göddeke, D.; Strzodka, R.; Mohd-Yusof, J.; McCormick, P.; Wobker, H.; Becker, C.; and Turek, S.: Using GPUs to Improve Multigrid Solver Performance on a Cluster. *Int. J. Comput. Sci. Eng.*, vol. 4, no. 1, 2008, pp. 36–55.

12. Göddeke, D.; Buijssen, S. H.; Wobker, H.; and Turek, S.: GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver. *High Performance Computing and Simulation 2009*, Leipzig, Germany, June 2009.

13. Göddeke, D.; and Strzodka, R.: Cyclic Reduction Tridiagonal Solvers on GPUs Applied to Mixed Precision Multigrid. *IEEE T. Parall. Distr.*, March 2010. Special Issue: High Performance Computing with Accelerators.

14. Phillips, E. H.; Zhang, Y.; Davis, R. L.; and Owens, J. D.: Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units. *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, Orlando, Florida, USA, January 2009.

15. Jacobsen, D.; Thibault, J.; and Senocak, I.: An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters. *48th AIAA Aerospace Sciences Meeting*, Orlando, Florida, USA, January 2010.

16. Anderson, W.; and Bonhaus, D.: An Implicit Upwind Algorithm for Computing Turbulent Flows on Unstructured Grids. *Comput. Fluids*, vol. 23, no. 1, 1994, pp. 1–21.

17. Park, M.; and Carlson, J.-R.: Turbulent Output-Based Anisotropic Adaptation. *AIAA-2010-0168*, January 2010.

18. Nielsen, E.; Diskin, B.; and Yamaleev, N.: Discrete Adjoint-Based Design Optimization of Unsteady Turbulent Flows on Dynamic Unstructured Grids. *AIAA J.*, vol. 48, no. 6, 2010, pp. 1195–1206.

19. Roe, P.: Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes. *J. Comp. Phys.*, vol. 43, no. 2, 1981, pp. 357–372.

20. Spalart, P.; and Allmaras, S.: A One-Equation Turbulence Model for Aerodynamic Flows. *Rech. Aerospatiale*, vol. 1, 1994, pp. 5–21.

21. Cuthill, E.; and McKee, J.: Reducing the Bandwidth of Sparse Symmetric Matrices. *Proceedings of the 24th Nat. Conf. ACM*, 1969, pp. 157–172.

22. Karypis, G.; and Kumar, V.: Multilevel Algorithms for Multi-Constraint Graph Partitioning. *Proceedings of the 1998 ACM/IEEE SC98 Conference*, 1998.

23. Karypis, G.; and Kumar, V.: Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distrib. Comput.*, vol. 48, no. 1, 1998, pp. 96–129.

24. Lee-Rausch, E.; Frink, N.; Mavriplis, D.; Rausch, R.; and Milholen, W.: Transonic drag prediction on a DLR-F6 transport configuration using unstructured grid solvers. *Comput. Fluids*, vol. 38, 2009, pp. 511–532.

| REPORT DOCUMENTATION PAGE | *Form Approved OMB No. 0704-0188* |
|---|---|

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
**PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

**1. REPORT DATE** *(DD-MM-YYYY)*: 01- 10 - 2012

**2. REPORT TYPE**: Technical Memorandum

**3. DATES COVERED** *(From - To)*: May 2010 - August 2010

**4. TITLE AND SUBTITLE**
Production Level CFD Code Acceleration for Hybrid Many-Core Architectures

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Duffy, Austen C.; Hammond, Dana P.; Nielsen, Eric J.

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**
877868.02.07.07.03.01.02

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
NASA Langley Research Center
Hampton, VA 23681-2199

**8. PERFORMING ORGANIZATION REPORT NUMBER**
L-20136

**9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
National Aeronautics and Space Administration
Washington, DC 20546-0001

**10. SPONSOR/MONITOR'S ACRONYM(S)**
NASA

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**
NASA/TM-2012-217770

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified - Unlimited
Subject Category 59
Availability: NASA CASI (443) 757-5802

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
In this work, a novel graphics processing unit (GPU) distributed sharing model for hybrid many-core architectures is introduced and employed in the acceleration of a production-level computational fluid dynamics (CFD) code. The latest generation graphics hardware allows multiple processor cores to simultaneously share a single GPU through concurrent kernel execution. This feature has allowed the NASA FUN3D code to be accelerated in parallel with up to four processor cores sharing a single GPU. For codes to scale and fully use resources on these and the next generation machines, codes will need to employ some type of GPU sharing model–as presented in this work. Findings include the effects of GPU sharing on overall performance. A discussion of the inherent challenges that parallel unstructured CFD codes face in accelerator-based computing environments is included, with considerations for future generation architectures. This work was completed by the author in August 2010, and reflects the analysis and results of the time.

**15. SUBJECT TERMS**
Computational fluid dynamics; Computer architecture; Parallel algorithms

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | STI Help Desk (email: help@sti.nasa.gov) |
| | | | | | 19b. TELEPHONE NUMBER *(Include area code)* |
| U | U | U | UU | 22 | (443) 757-5802 |

**Standard Form 298** (Rev. 8-98)
Prescribed by ANSI Std. Z39.18