1-4-2011

A Full-Depth Amalgamated Parallel 3D Geometric Multigrid Solver for GPU Clusters

Dana A. Jacobsen
Boise State University

Inanc Senocak
Boise State University

49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition
4 - 7 January 2011, Orlando, Florida

AIAA 2011-946

A Full-Depth Amalgamated Parallel 3D Geometric Multigrid Solver for GPU Clusters

Dana A. Jacobsen*, and Inanc Senocak†
Boise State University, Boise, Idaho, 83725

Numerical computations of incompressible flow equations with pressure-based algorithms necessitate the solution of an elliptic Poisson equation, for which multigrid methods are known to be very efficient. In our previous work we presented a dual-level (MPI-CUDA) parallel implementation of the Navier-Stokes equations to simulate buoyancy-driven incompressible fluid flows on GPU clusters with simple iterative methods while focusing on the scalability of the overall solver. In the present study we describe the implementation and performance of a multigrid method to solve the pressure Poisson equation within our MPI-CUDA parallel incompressible flow solver. Various design decisions and algorithmic choices for multigrid methods are explored in light of NVIDIA's recent Fermi architecture. We discuss how unique aspects of an MPI-CUDA implementation for GPU clusters is related to the software choices made to implement the multigrid method. We propose a new coarse grid solution method of embedded multigrid with amalgamation and show that the parallel implementation retains the numerical efficiency of the multigrid method. Performance measurements on the NCSA Lincoln and TACC Longhorn clusters are presented for up to 64 GPUs.

I. Introduction

Graphics processing units (GPUs) have enjoyed rapid adoption within the high-performance computing (HPC) community. GPUs have evolved from hardware rendering pipelines that were not amenable to non-rendering tasks, to the modern General Purpose Graphics Processing Unit (GPGPU) paradigm. GPU clusters, where fast network connected compute-nodes are augmented with latest GPUs,[1] are now being used to solve challenging problems from various domains. Examples include the 384 GPU Lincoln Tesla cluster deployed in February 2009 by the National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana Champaign[2], the 512 GPU Longhorn cluster deployed in January 2010 by the Texas Advanced Computing Center (TACC)[3], the 680 GPU TSUBAME 1.2 cluster deployed in October 2008 at the Tokyo Institute of Technology[4]. These successful installations and performance demonstrations[4–7] paved the way for petascale supercomputers powered by GPUs. As of December 2010, three of the world's top five supercomputers (7168-GPU Tianhe-1A[8], 4640-GPU Nebulae[9], 4224-GPU Tsubame 2.0[10]) on the Top 500 systems are now powered by NVIDIA GPUs. Power consumption of supercomputers is increasingly becoming a concern. These three GPU-accelerated supercomputers also stand out with their lower power consumption. relative to computational performance.

Owens et al.[11] survey the early history as well as the state of GPGPU computing up to 2007. Kindratenko et al.[12] describe applications, developed for GPU clusters, in cosmology, molecular dynamics, and quantum chemistry, among others. In the computational fluid dynamics field (CFD) both Euler and Navier-Stokes solvers have been developed for GPU computing platforms[5,13–19]. Jacobsen et al.[5] and Thibault and Senocak[18] provide a detailed account of GPU computing in the CFD field.

Early efforts to develop numerical applications for medium sized GPU clusters have been promising, but with the introduction of petascale supercomputers that are powered by GPUs, a major challenge ahead of the scientific computing community is to extend advanced numerical algorithms beyond a single GPU. A good example is the multigrid method to solve boundary value problems such as the Poisson equation. They are described briefly in Press et al.[20] and in detail in Briggs[21] and in Trottenberg et al.[22]. The fundamental idea is to apply multiscale techniques, where the problem is solved at multiple resolution levels. This leads to not only a very efficient method with excellent

*Graduate Research Assistant, Department of Computer Science, Student Member AIAA.
†Assistant Professor, Department of Mechanical & Biomedical Engineering, Senior Member AIAA.

convergence, but one where the convergence rate can be independent of the grid size, which potentially makes the multigrid method a truly scalable algorithm for parallel computing.

Parallel multigrid solvers have been studied for some time. McBryan et al.[23] survey parallel multigrid methods, and Chow et al.[24] give a more recent survey of important techniques. Particularly relevant to this work is the discussion on domain partitioning and the discussion of methods for coarse grid solving. McBryan et al. show parallel efficiency results for numerous architectures, indicating very poor weak scaling results with most implementations. In contrast, the very recent results of Göddeke[25] show very good scaling on a GPU cluster, albeit with comparatively slow NVIDIA FX1400 graphics cards.

Previous studies using multigrid for GPUs include Goodnight et al.[26] who describe a multigrid solver on a single GPU for boundary value problems. This early work was done using early GPU hardware that were limited in both hardware and software compared to current state of the art in GPU computing. Cohen and Molemaker[16] describe the single GPU implementation and validation of an incompressible Navier-Stokes solver with Boussinesq approximation, using a multigrid solver. Göddeke et al.[15] describe integrating parallel multigrid solvers into an existing finite element solver using mixed precision. This is done in a framework of multiple CPU and GPU solvers, with choices that are made dynamically at each step.

Göddeke[25] discusses many aspects of multigrid on GPUs and GPU clusters. Parallel multigrid is investigated as part of an unstructured finite element solver which runs on CPUs or GPUs. Of particular relevance to this work is his discussion of smoothers, coarse grid solvers, multigrid cycle type, and general comments on GPU and GPU cluster performance. The overall structure of the solver is quite different from that used in this study, as are the approaches taken for using parallelism at different multigrid levels.

Thibault and Senocak[18] developed a single-node multi-GPU 3D incompressible Navier-Stokes solver with a Pthreads-CUDA implementation that targets multi-GPU desktop platforms. This work was extended in Jacobsen et al.[5] where an MPI-CUDA implementation was presented and assessed on the NCSA Lincoln Tesla Cluster. These papers give details on the software implementation for multi-GPU clusters. However, a fixed iteration Jacobi method was used to solve the pressure Poisson equation for simplicity to avoid software complexity while developing a multi-GPU CFD solver. The present work builds upon these early efforts and incorporates a geometric multigrid solver to solve the pressure Poisson equation. We describe the implementation, software design decisions, and parallel performance of a multigrid method to solve the pressure Poisson equation within a 3D incompressible flow solver designed for GPU clusters.[5] We describe the overall changes made to the underlying flow solver as well as the various design decisions for multigrid solvers, with reasoning given for the choices and tradeoffs made. Special attention is paid to the impact of NVIDIA's recent Fermi architecture as well as the unique aspects of the dual-level MPI-CUDA programming model

II. Governing Equations and Numerical Approach

Navier-Stokes equations for buoyancy driven incompressible fluid flows can be written as follows:

$$\nabla \cdot \mathbf{u} = 0, \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho}\nabla P + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \tag{2}$$

where $\mathbf{u}$ is the velocity vector, P is the pressure, ρ is the density, ν is the kinematic viscosity, and $\mathbf{f}$ is the body force. The Boussinesq approximation, which applies to incompressible flows with small temperature variations, is used to model the buoyancy effects in the momentum equations[27]:

$$\mathbf{f} = \mathbf{g} \cdot (1 - \beta(T - T_\infty)), \tag{3}$$

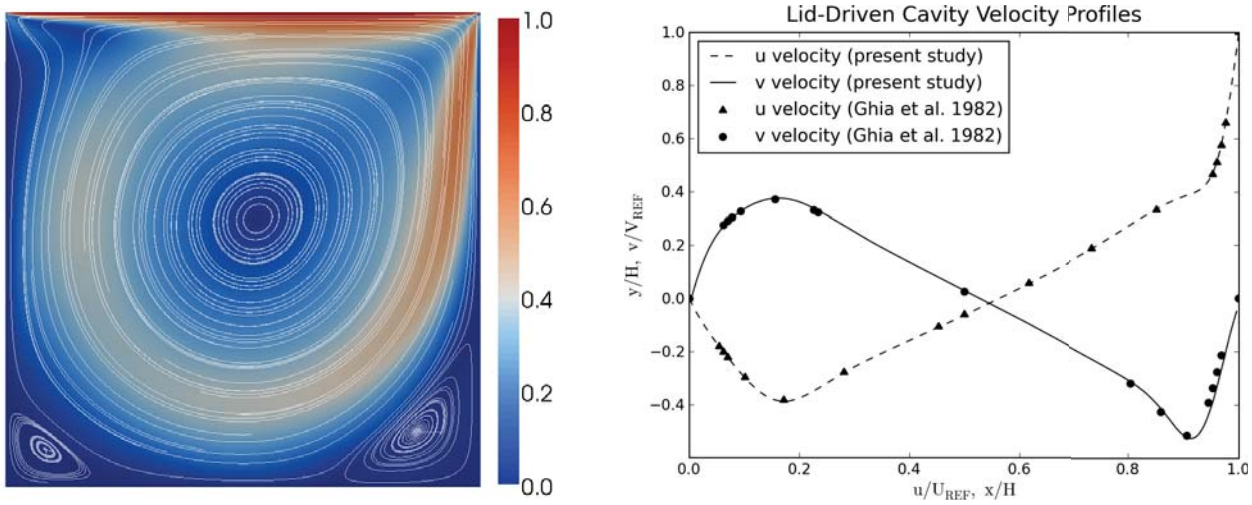
where $\mathbf{g}$ is the gravity vector, β is the thermal expansion coefficient, T is the calculated temperature at the location, and T_∞ is the steady state temperature.

The temperature equation can be written as[28,29]

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \alpha \nabla^2 T + \Phi, \tag{4}$$

where α is the thermal diffusivity and Φ is the heat source.

The above governing equations are discretized on a uniform Cartesian staggered grid with second order central difference scheme for spatial derivatives and a second order accurate Adams-Bashforth scheme for time derivatives. The



a) b)

Figure 1. Lid-driven cavity simulation with $\mathrm{Re} = 1000$ on a $257 \times 33 \times 257$ grid. 3D computations were used and a 2D center slice is shown. a) Velocity streamlines and velocity magnitude distribution. b) Comparison to the benchmark data from Ghia et al.[31].

projection algorithm[30] is then adopted to find a numerical solution to the Navier-Stokes equation for incompressible fluid flows, which requires the solution of a Poisson equation for pressure that can be written as

$$\nabla^2 P^{t+1} = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u}^*. \tag{5}$$

where P^{t+1} is the pressure at timestep $t+1$, ρ is the density, and $\mathbf{u}^*$ is the predicted velocity field.

This pressure Poisson equation (Eq. (5)) is solved using a parallel geometric multigrid method which is described in the following sections. The present implementation supports double precision on NVIDIA GPUs with compute capability 1.3 or higher. We observe a performance penalty approximately $2-3\times$ relative to single precision computations. Most performance results in this paper are shown using double precision to allow residual differences to be computed to very low levels.

Validation on a number of test cases including the well-known lid-driven cavity and natural convection in heated cavity problems[31,32] were used to compare the overall solutions to known results. Figure (1) presents the results of a lid-driven cavity simulation with a Reynolds number 1000 on a $257 \times 33 \times 257$ grid. Figure (1a) shows the velocity magnitude distribution and streamlines at mid-plane. As expected, the computations capture the two corner vortices at steady-state. In Fig. (1b), the horizontal and vertical components of the velocity along the centerlines are compared to the benchmark data of Ghia et al.[31]. The results agree well with the benchmark data. While the multigrid implementation described in this paper has been used for a variety of simulation types, results shown are using this benchmark lid-driven cavity model.

III. Geometric Multigrid Method

Direct solution of large systems of partial differential equations is computationally overwhelming, both in time and space. Iterative solvers such as the Jacobi and Gauss-Seidel methods offer a practical alternative, however their convergence rate can be quite slow for large domains – a fact noted by Seidel in 1874. The conjugate gradient method (CG) discovered independently in 1952 by Hestenes and Stiefel, with extensions by Lanczos, provides a much faster technique, though still proportional to the domain size. The works by Fedorenko (1962) and Bakhvalov (1966) were the first to investigate multigrid techniques, showing their asymptotic optimality, the extensive work by Brandt in the 1970s[33,34] demonstrated the numerical efficiency of the method in operation, as well as developing many of the ideas used in current techniques.

Given the system $Au = f$ where A is an $N \times N$ matrix, the solver starts with an initial guess v. As iterative solvers proceed, the current solution v converges to the real solution u. From this, the error

$$e = u - v, \tag{6}$$

and the residual

$$r = f - Av, \tag{7}$$

can be defined. Manipulation of these give rise to the residual equation $Ae = r$ and the residual correction $u = v + e$. Together these formulate the residual correction algorithm:

$$\begin{array}{ll} \textsf{begin} & \\ \quad r = f - Av & \text{calculate residual} \\ \quad e = \textsf{Solve}(A, r) & \text{solve for the error} \\ \quad u = v + e & \text{residual correction} \\ \textsf{end} & \end{array} \tag{8}$$

which shows how solving for the error can solve the equation. The multigrid method will make use of this idea.

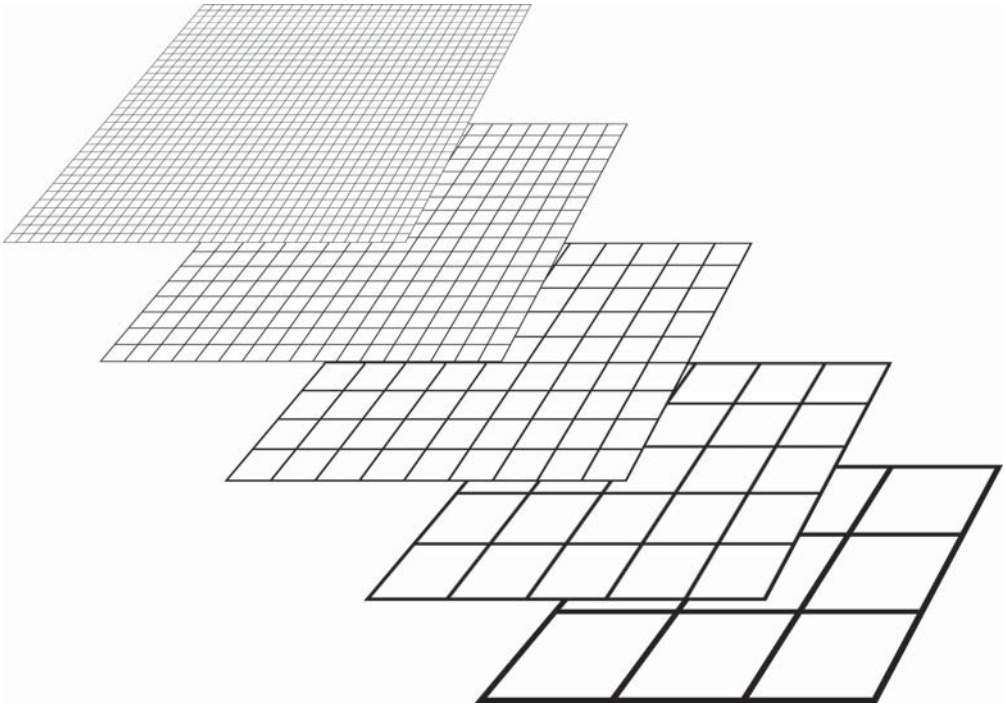
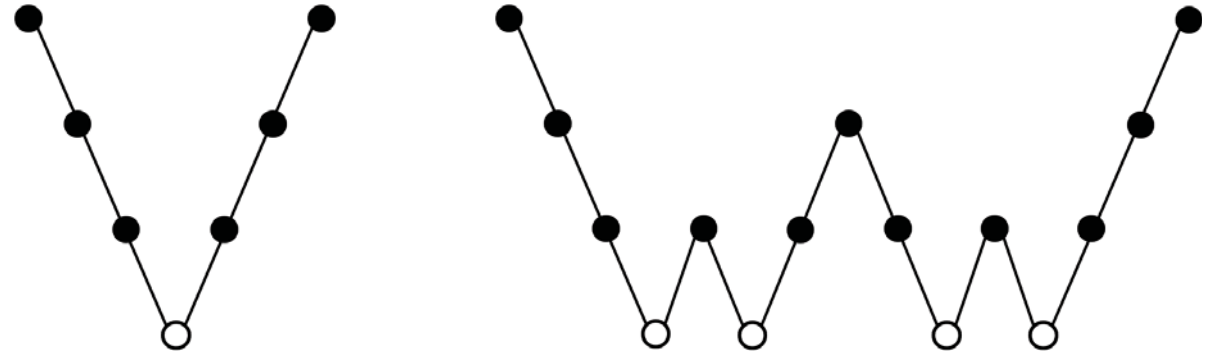


Figure 2. A view of a grid hierarchy with a structured rectilinear grid, going from 33^3 at the fine level to 3^3 at the coarsest level.

When running an iterative solver such as Jacobi or Gauss-Seidel, an observation which can be made is that the error is *smoothed*. High frequency terms in the error rapidly diminish, while low frequency terms are removed much more slowly. This is a natural outcome of the narrow support of the discrete operation, where values can only propagate one grid cell per iteration. The multigrid method makes use of this behavior, and these are called *smoothers* when run inside multigrid.

Given a smoothed error matrix, looking at the error on a coarser grid would raise the frequencies — low frequency errors on the fine grid become high frequency errors on the coarse grid. This leads to multigrid: errors are smoothed at the current grid level, the residuals are transferred to a coarser grid by a process called *restriction*, the problem is further solved at this coarse level, and then the result is interpolated back to the current grid (known as *prolongation*). This process creates a hierarchy of grids (see example in Fig. (2)), with the coarsest grid being solved by another

method such as a direct solver or a stationary iterative technique such as Jacobi or Gauss-Seidel. The multigrid cycle may be repeated, and it does not have to follow a strict progression of fine to coarse. Two example cycle types, the common V-cycle and W-cycle, are represented in Fig. (3).



a) V-cycle b) W-cycle

Figure 3. Examples of multigrid cycle types. Closed circles represent smoothing and open circles represent a coarsest grid solve (which may be approximate). a) V-cycle, b) W-cycle.

```
proc mgcycle(γ, u_k, f, s_pre, s_post)
  for i := 1 to s_pre do : Smooth                          presmoothing
  r_k := f − L u_k                                         compute residual using Laplacian
  r_{k−1} := R r_k                                         restrict residual
  if k − 1 = coarsest level
   then
        CoarseGridSolve(u, f, r_{k−1})                     Apply coarse grid solver
   else
        for i := 1 to γ do :
            mgcycle(γ, 0, r_{k−1}, s_pre, s_post)          multigrid on coarser levels
  fi
  u_k := u_k + P v_{k−1}                                   prolong coarse result to this level
  for i := 1 to s_post do : Smooth                         postsmoothing
```

Figure 4. Geometric Multigrid Algorithm

A distinction can be made between geometric multigrid methods, which define coarse meshes directly from the fine mesh, and algebraic methods which operate directly on the matrix of equations. The algebraic method is usually preferable for complex or unstructured grids. Since this study uses a structured Cartesian mesh, the geometric multigrid method is used. The complete algorithm for a multigrid cycle is shown in Fig. (4). Setting $\gamma = 1$ will give a V-cycle, while $\gamma = 2$ gives a W-cycle.

IV. MPI-CUDA Implementation of the Parallel Multigrid Method

Multiple programming APIs (Application Programming Interfaces) along with a domain decomposition strategy for data-parallelism is required to achieve high throughput and scalable results from a CFD model on a multi-GPU platform. When more than one GPU is used, cells at the edges of each GPU's computational space must be communicated to the GPUs that share the domain boundary so they have the current data necessary for their computations. Data transfers across the neighboring GPUs inject additional latency into the implementation which can restrict scalability if not properly handled.

CUDA is the API used by NVIDIA for their GPUs.[35] CUDA programming consists of kernels that run on the GPU and are executed by all the processor units in a SIMD (Single Instruction Multiple Data) fashion. The CUDA API also extends the host C API with operations such as `cudaMemcpy()` which performs host/device memory transfers. Memory transfers between GPUs on a single host are done by using the host as an intermediary – there are no CUDA commands to operate between GPUs. On a given thread, CUDA kernel calls are asynchronous (i.e. control is given back to the host CPU before the kernel completes) but do not overlap (i.e. only one kernel runs at a time). Memory operations are synchronous and do not start until previous kernels have completed unless the CUDA streams functionality is used, which provides a mechanism for memory operations to run concurrently with kernel execution as well as host computation.

The Message Passing Interface (MPI) API[36] is widely used for programming clusters, and works on both shared and distributed memory machines while providing a highly portable solution to writing programs to work on a wide variety of machines and hardware topologies. Jacobsen et al.[5] have shown that an MPI-CUDA implementation can perform well on both multi-GPU workstations and GPU clusters. This dual-level MPI-CUDA approach is adopted for the implementation, and details about the GPU programming using CUDA as well as the methods used for MPI communication and domain decomposition can be found in that paper. MPI is used to map each GPU to a process, and is used for inter process communication. Three degrees of computation / communication overlapping have been implemented, allowing study of this cluster programming aspect. Domain decomposition is in 1D across GPUs which leads to optimal host / GPU memory transfers, though suboptimal network communication sizes. An orthogonal 2D decomposition is performed inside the GPUs using CUDA as the parallelization mechanism.

A. GPU Implementation

Listing 1 shows the host code for a multigrid cycle. The `mgdata_t` structure array is initialized only once, and contains parameters for each level (grid sizes and spacing, pointers to pressure and residual at each level, etc.). Setting $gamma = 1$ leads to a V-cycle, while $gamma = 2$ leads to a W-cycle. Four routines are called from this host code: restriction, prolongation, a smoother, and a method for the coarsest grid solve. The `MGzeroMesh` function is a wrapper around `cudaMemset` to zero the appropriate device memory.

1. Restriction

The restriction step calculates the residual for the fine grid and downscales (restricts) it to a coarser grid. This is done as two distinct kernels, where the first uses a Laplacian operation to calculates the fine grid residual values. After a neighbor exchange to update the ghost cells, a restriction kernel is run over the coarse grid. The restriction operation takes a weighted average of neighboring cells on the fine grid. Common methods include 1-point injection, 7-point half-weighting, and 27-point full-weighting.

On isotropic simulations, where $\Delta X = \Delta Y = \Delta Z$, half-weighting performs quite well. The convergence rate drops significantly as the simulation becomes anisotropic – where one dimension is significantly different from another. Full-weighting maintains the best convergence rates in these situations. In our implementation, the restriction kernel runs over the coarse grid, hence operations with eight times fewer threads than a fine-grid kernel. Partly due to this factor, the increase in global convergence rate far outweighs the small amount of extra time taken for full weighting. The 27-point full-weighting is used for all results in this paper.

2. Prolongation

The prolongation operation for multigrid ought to be the inverse operator of the restriction operation. With full-weighting used for restriction, the 3D adjoint is trilinear interpolation. The value in the new fine grid will be an distance-weighted average of the values of the surrounding coarse grid cells. Care must be taken with the operation ordering to prevent undesired floating point rounding differences. Similar to the restriction kernel, this operation can be made to run over the coarse grid.

3. Smoother

Two smoothers have been implemented: weighted Jacobi and Red-Black Gauss-Seidel, including a relaxation parameter to allow overrelaxation. The host code for weighted Jacobi performs a number of iterations of the sequence: smoother kernel, exchange, set boundary conditions. Additionally an exchange is done at the end to ensure all ghost

```
static void mgcycle (mgdata_t* mgd,
                     int m, int mgend,
                     int gamma,
                     REAL* dphibuf, mpi_exchange* mex)
{
   int g;
   assert(gamma > 0);
   assert(m < mgend);

   // Smooth the result
   //   : smooth E[m] using initial value and R[m]
   MG_SMOOTHER(mgd, m, HCONSTANT_SMG_ITERV1, dphibuf, mex);

   // Make a coarser mesh of residuals.
   //   : E[m] and R[m] create R[m+1]
   restriction(mgd, m, dphibuf, mex);

   // Clear out the initial corrections for the coarser level
   //   : E[m+1] = 0
   MGzeroMesh(mgd, m+1);

   if (m+1 >= mgend) {
      mg_coarse_grid_solve(mgd, m+1, dphibuf, mex);
   } else {
      for (g = 0; g < gamma; g++) {
         mgcycle(mgd, m+1, mgend, gamma, dphibuf, mex);
      }
   }

   // Create the finer mesh
   //   : E[m] += interpolated E[m+1]
   prolongation(mgd, m+1, mex);

   // Smooth the result
   //   : smooth E[m] using initial value and R[m]
   MG_SMOOTHER(mgd, m, HCONSTANT_SMG_ITERV2, dphibuf, mex);
}
```

Listing 1. The host code for a multigrid cycle.

cells are consistent. The Jacobi smoother allows computation / communication overlap. The host code for weighted Red-Black Gauss-Seidel performs a number of iterations of the sequence: smoother kernel (red), exchange, smoother kernel (black), set boundary conditions, exchange.

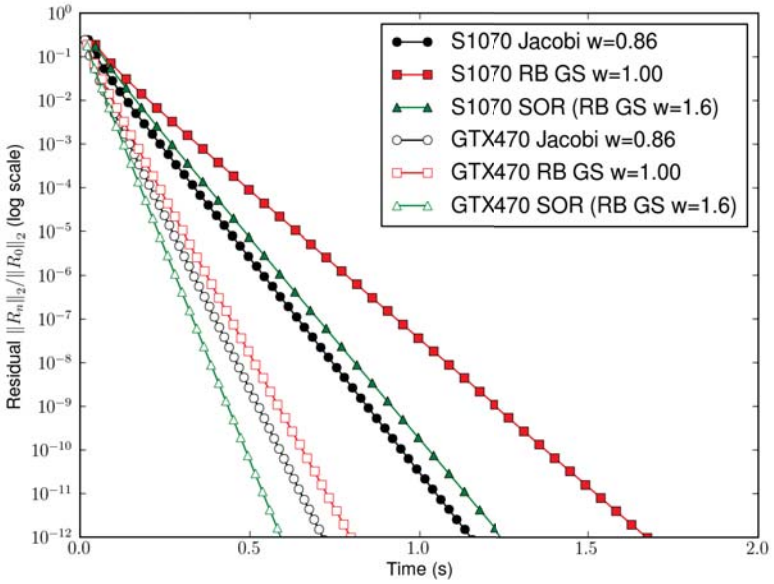


Figure 5. Comparison of smoothers used inside a multigrid V-cycle on the NVIDIA Tesla S1070 and GTX 470 with the Fermi architecture. Time is plotted against the residual level for a 129^3 problem on a single GPU using double precision calculations. 4 pre-smoothing and 4 post-smoothing iterations at each grid level. No multigrid truncation was applied. The weighted Jacobi solver uses shared memory.

Figure (5) shows the performance of the smoothers on two platforms when used inside a multigrid V-cycle. The time taken counts all multigrid activity, of which the smoother was measured to be between 67% and 82% of the total. As expected, the spacing between cycles indicates the the Gauss-Seidel method converges faster than weighted Jacobi ($\omega = 0.86$) and the SOR weighting ($\omega = 1.60$) converges faster yet. On both platforms using weighted Jacobi finishes faster than Red-Black Gauss-Seidel, which indicates an implementation difference. Using SOR with a near-optimal weight for this problem, the performance improves substantially, but does not outweigh the implementation performance difference on the S1070. It does lead to the fastest solution on the GTX 470. Comparing the two architectures – the previous generation Compute Capability 1.3 S1070 and the current Compute Capability 2.0 (Fermi) GTX 470, not only is the Fermi system faster overall, but all the solutions are closer.

4. Coarse Grid Solve

Classic multigrid uses a full-depth cycle along with an exact solution for the coarsest grid. When data is distributed among multiple nodes of a parallel system, it is not possible to reduce the depth to the lowest level without repartitioning the data. Additionally, if the grid size is not identical in all dimensions, some form of semi-coarsening (restriction in only some of the dimensions) is needed to continue reduction below the point where the smallest dimension has only one interior cell. Finally, even when this reduction is done, the varying boundary conditions allowed make an exact solution not as simple as for the case of all-Dirichlet boundaries.

The first method applied is the approximate solution method, where a fixed number of iterations of the smoother are applied at the coarsest level. Figure (6) shows the effect of of this method as the multigrid cycle is truncated earlier. While excellent convergence and computational performance is obtained with deep cycles, with earlier truncation the performance begins to take on the characteristics of the iterative solver.

Another method tried is a dynamic number of iterations of the smoother, where the number of iterations is either

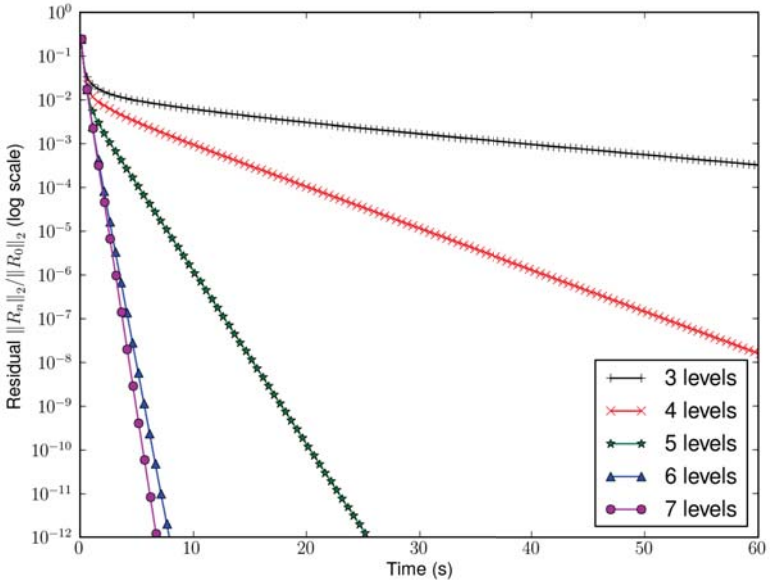


Figure 6. Effect of early truncation level using V-cycles. Time is plotted against the residual level for a 257^3 problem on a single Tesla S1070 GPU using double precision calculations. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. The coarse grid solver is 16 iterations of the weighted Jacobi smoother. A marker is shown for each 4 loops of the multigrid cycle. The coarsest grid in this example for 3 levels is 65^3, and for 7 levels is 5^3.

based on the coarsest grid size or is run until the residual is reduced to a desired accuracy level. This mitigates the effect of cycle truncation on convergence, but the number of iterations required grows quite rapidly. Over 1,000 iterations of the Jacobi solver are needed to obtain an accurate solution on a 65^3 grid. This is even less acceptable on a cluster, where communication must be performed between each iteration. One solution to this dilemma is to use a well-tuned parallel solver, such as parallel conjugate gradient. This is the solution used by Göddeke[25], where a conjugate gradient coarse grid solver is used and set to reduce the initial residuals by two decimal digits.

The solution we propose is to use multigrid as the coarse grid solver. This embedded multigrid solver could use different parameters (smoother type, number of iterations, cycle type) than the outer multigrid solver and could be iterated more than once. While on a single GPU this method has little impact (if the parameters are identical, it is identical to performing no truncation), it has some strong advantages on a cluster. These will be explored in section III.C.

B. Computational Overlapping

After performing a an operation over the entire domain, each process must perform an exchange operation, where the data in neighboring cells is communicated. Overlapping of communication and computation during these exchanges is critical to performance[5,7,37,38]. We have implemented three overlapping strategies which are described in detail in the author's 2010 AIAA paper[5]. These are not the primary focus of this paper, but are investigated briefly in light of the multigrid implementation.

The three methods are (1) No computational overlapping, which uses non-blocking MPI calls to alleviate the effect of the GPU memory transfer but does not overlap the GPU computation, (2) Simple overlapping which computes and transfers the GPU edge data, then overlaps the MPI communication with GPU computation of the center, and (3) Fully overlapped, which uses the CUDA streams functionality to asynchronously perform GPU computation, GPU/host memory transfers, and MPI network communication. When not specified, the fully overlapped version is used for results in this paper.

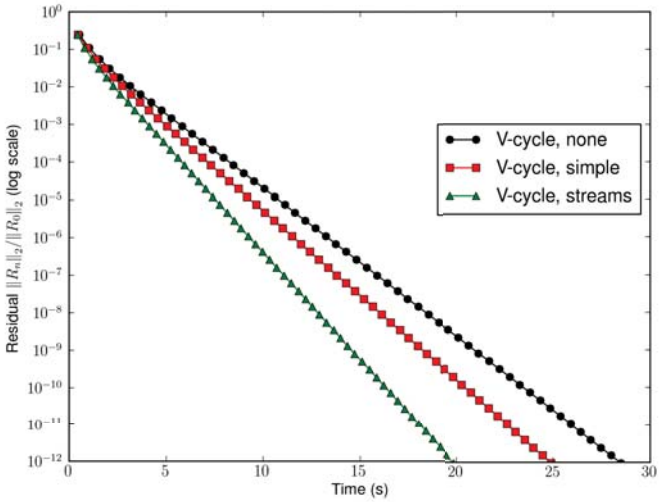
The three overlapping strategies are also used in the smoother and restriction operations during the multigrid process. The prolongation operation does not overlap, but it uses less than 10% of the multigrid time on these tests. Figure (7) compare the three strategies in the multigrid solver for a calculation using eight GPUs (four nodes of the NCSA Lincoln Tesla cluster). With V-cycles, the performance gain from overlapping is significant for both simple overlapping and again when using CUDA streams. Both show an improvement over the W-cycle results for this problem, and benefits for overlapping are minimal for these W-cycles results.

C. Amalgamated Multigrid for GPU Clusters

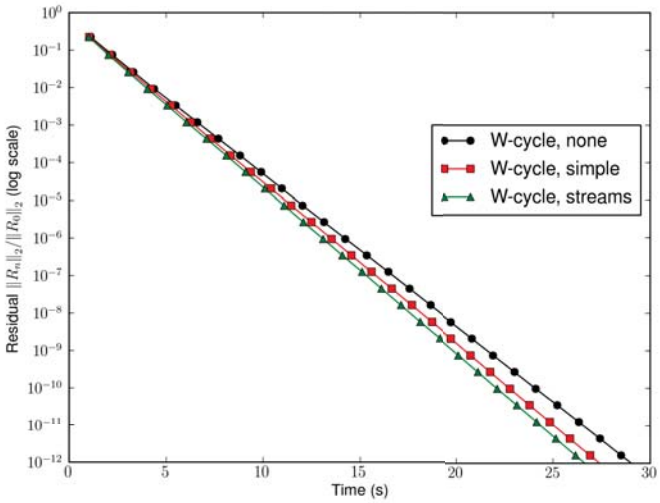
In a parallel multigrid implementation, the decomposition of each coarsest grid is a critical factor. In this implementation, the dimensions (nx_c, ny_c, nz_c) of the coarse grid are $nx_c = (nx_f - 1)/2 + 1$, $ny_c = (ny_f - 1)/2 + 1$, and $nz_c = (nz_f - 1)/2 + 1$. No semi-coarsening is applied, meaning each dimension is reduced. Each GPU is responsible for the coarse grid pixels deriving from the restriction operator applied to its current domain, meaning the partitioning is static on each grid. One implication of this partitioning is that the number of GPUs used remains constant while the work decreases by a factor of $2^3 = 8$ at each step. More importantly, when $nz_c = \#gpus$, no more coarsening can be applied, as this would result in some GPUs having no pixels. Even before this time, the amount of work on each GPU is very small.

To alleviate the issue of rapidly shrinking work and the effect it has on computation / communication ratios, it is not uncommon for the multigrid cycle (e.g. V-cycle) to be truncated, and the coarsest grid solver be set to an algorithm such as parallel conjugate gradient as used in Göddeke[25]. A related idea is discussed by Gropp[39] and expanded on by Chow et al.[24], which is to amalgamate the grids at the coarse level, meaning a gather operation combines the coarse grids at all levels, which is then solved on a single node, and the results then scattered back. Since the coarse levels are extremely small compared to the fine grid, the amount of data distributed across the network is relatively small, and the resulting combined coarse grid level may be more effectively solved.

Gropp further notes that by using an allgather operation and redundant calculations, each node can calculate the coarse grid, which means there is no need for a scatter operation. The allgather method was implemented and compared with the regular gather/solve/scatter solution. Measured times were effectively equal at most truncation levels, but were longer with shallow truncation levels (where the coarse grid was large). While in theory the performance of `MPI_Allgather` can approach that of `MPI_Gather`, tests on the NCSA Lincoln Tesla cluster show that



a)



b)

Figure 7. Comparison of overlapping strategies. Time is plotted against the residual level for a double precision 513^3 problem using 8 GPUs. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. Truncation occurs at 6 levels (17^3), where the coarsest grid is amalgamated to a single GPU and four V-cycles are performed on this grid. A marker is shown for each loop of the multigrid cycle. a) V-cycle, b) W-cycle

`MPI_Allgather` is slower, and the difference widens as more GPUs are added. With 8 nodes it is 1.5 to 2 times slower, which confirms the results. An additional consideration for large clusters is power consumption, which may mean large clusters will want to reduce redundancy when it offers no clear benefit. Future directions for GPU scheduling may also allow GPUs to be dynamically used by other users, which indicate that the amalgamation technique of idling GPUs at small problem sizes may present a benefit to the total cluster throughput.

Altogether, the issue of coarse grid solution is one of solving the fine grids with high performance, where domains are large and performance is dominated by computation and bandwidth dominate, and also effectively solving coarse grids, where domains are small and performance is dominated by latency. There are numerous ways to approach the solution, and a number of enhancements can be used for each. What this study proposes is an *embedded multigrid with amalgamation* strategy.

For the coarsest grid solve, it is clear from Fig. (6) that a process of truncation with Jacobi iterations is not acceptable, as many GPUs will force early truncation. Amalgamating the coarse grids of GPUs allows deeper levels to be taken. In the limit, this means amalgamation to a single GPU where a full depth multigrid cycle can be performed. Hence, multigrid is embedded as the coarse grid solver in the parallel multigrid implementation. Convergence rates are now equal to those seen with textbook full-depth multigrid cycles, and for large coarse grids (e.g. $129 \times 129 \times 129$ seen on 128 GPU tests), performance is high relative to other methods. Since network communication costs are zero within the amalgamated solver, it can be advantageous to perform multiple multigrid cycles at the coarse level, leading to an improvement in the overall convergence rate for very little cost.

Amalgamation can often be performed with little to no extra memory use, as the rapid reduction in size per coarsening quickly reduces the memory needed. The current implementation can amalgamate to a single GPU at 3 levels for most problems without using any additional memory, as the second pressure buffer used by the Jacobi solver can be split into three parts holding the coarse grid pressure values, the coarse grid residuals, and a pressure buffer sized for the coarse grid.

One limitation of the current implementation are that a single depth is chosen, at which point all the results are amalgamated to a single GPU. While this drives communication costs to zero during the coarse solve, it also removes all parallelism from the other $N - 1$ GPUs. It is likely that a solution which fans in, for instance from 64 GPUs, to 8, then to 1, would better control the computation / communication ratio. Another limitation is that even on a single GPU, there is likely to be a point earlier than the coarsest $3 \times 3 \times 3$ grid where a different solver such as conjugate gradient will be faster.

V. Performance Results with NCSA Lincoln Tesla and TACC Longhorn Clusters

A comparison of the computational performance between multigrid and unweighted Jacobi is shown in Fig. (8). A 3D lid-driven cavity problem was started at three different grid sizes, and the time taken by the pressure solver is plotted against the residual level for the initial time step.

Figure (9) shows the performance of the multigrid algorithm on a GPU cluster for relatively large problems (16M, 128M, and 1024M cells). In particular the results of the amalgamation with embedded multigrid are compared to the fixed iteration approximate coarse grid solver. With 8 GPUs, the coarsest grid is 17^3, while with 64 GPUs with coarsest grid is 65^3. On a single GPU a full-depth V-cycle was performed, hence no truncation performed.

Figure (10) shows the GPU cluster performance at different amalgamation levels. Amalgamating at a shallow level leads to a very large coarse grid, for example in the 1025^3 problem using 3 multigrid levels, the coarsest grid is 257^3. This leads to poor performance both because of the large size to be communicated and for the loss of parallel computation at a level where it is still helpful. It can seen from the figure that there is a small benefit to amalgamating earlier than is required by the parallel implementation. Comparing the two clusters, the 2 GPU solution has little difference as it benefits only from the increased host/device memory bandwidth, while the 16 GPU solution is over 50% faster on the TACC Longhorn cluster, where the increased Infiniband bandwidth leads to faster performance.

Figures (11) and (12) break out the weak scalability by multigrid component. The prolongation component has the worst scaling, but also takes the least amount of time. Restriction scales better than the other components, but still shows poor scaling behavior at 16 GPUs. The smoother (weighted Jacobi for this case) takes the majority of the time, and shows scalability between the other components. While scaling to two GPUs is not bad, scaling to 16 GPUs is disappointing. Comparing the scaling of the two measured systems shows that the scaling to 16 GPUs is improved on the Longhorn system but still trends downward.

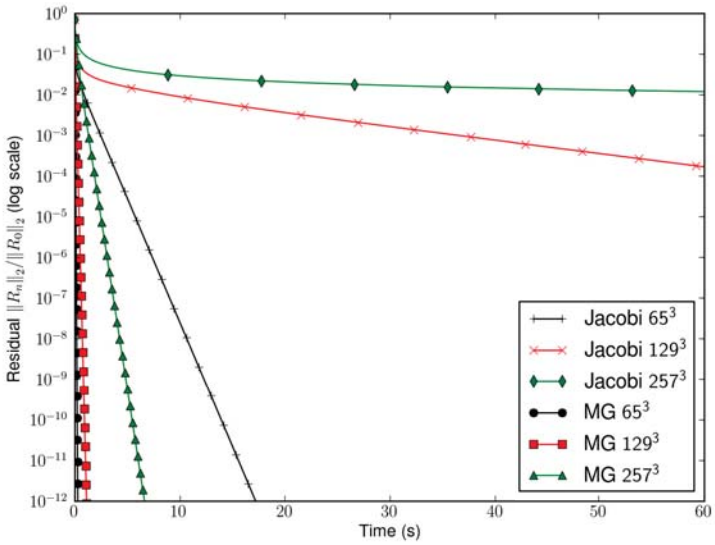


Figure 8. Performance of full-depth V-cycle multigrid compared to iterative Jacobi. Time is plotted against the residual level for a 65^3, 129^3, and 257^3 problem on a single Tesla S1070 GPU using double precision calculations. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. For clarity, a marker is shown for each 2 loops of the multigrid cycle, every 1000 Jacobi iterations at 257^3, and every 4000 Jacobi iterations at 129^3 and 65^3.

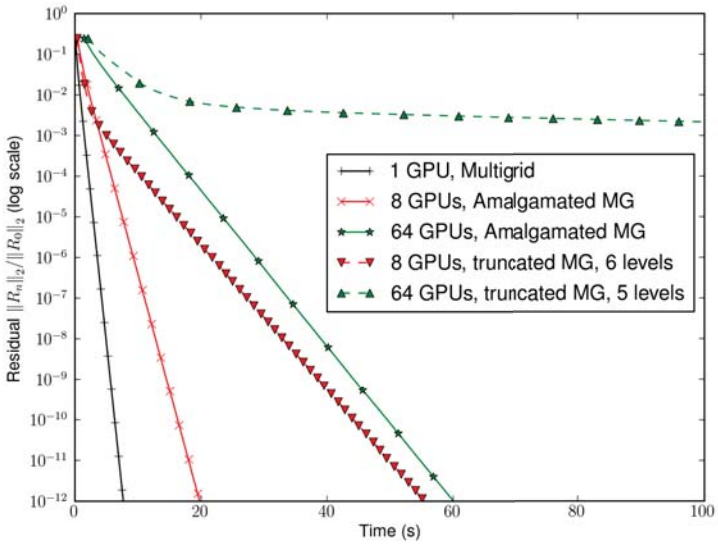


Figure 9. Convergence and parallel efficiency of truncated and amalgamated multigrid on 1, 8, and 64 GPUs where the problem size scales with the number of GPUs. Time is plotted against the residual level for a double precision problem using 257^3 on 1 GPU, 513^3 using 8 GPUs, and 1025^3 using 64 GPUs on the NCSA Lincoln Tesla cluster. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. A marker is shown for each 4 loops of the multigrid cycle. V-cycles and CUDA streams overlapping were used for each.

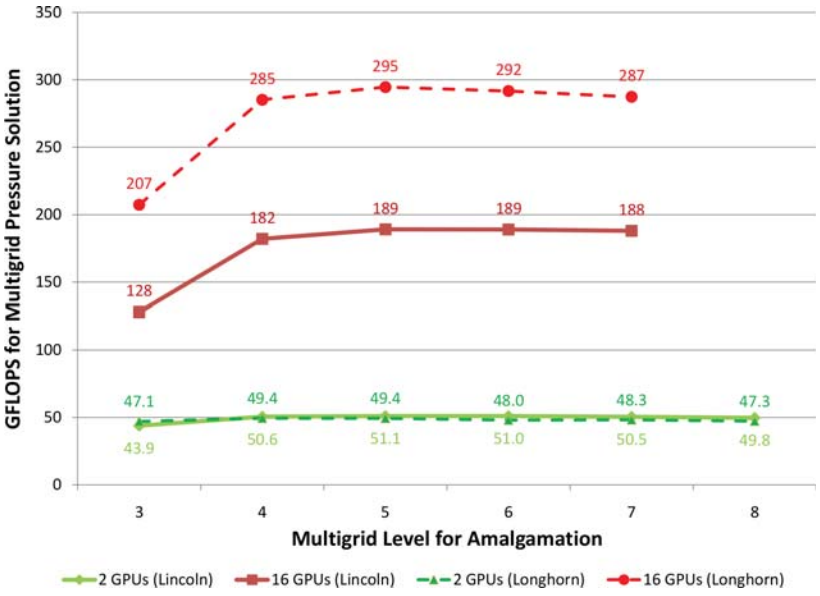


Figure 10. Performance of the multigrid solver with different amalgamation levels selected, using single precision. Problem size scales with the number of GPUs, with 513^3 on 2 GPUs and 1025^3 using 16 GPUs on the NCSA Lincoln Tesla and TACC Longhorn clusters. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. V-cycles and CUDA streams overlapping were used for each.

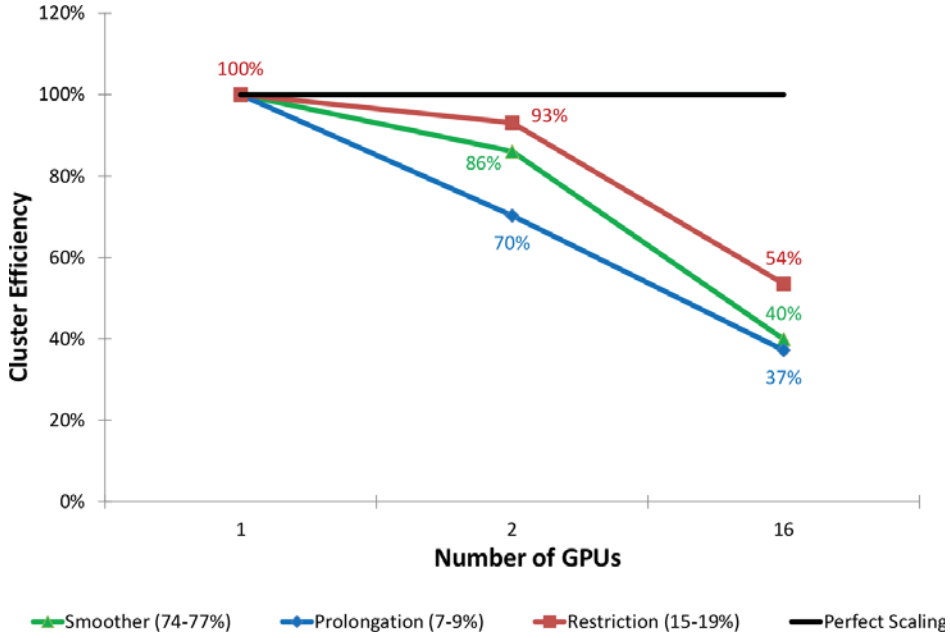


Figure 11. Weak scaling performance of the multigrid solver components, with their overall portion of the solver time shown. Amalgamated multigrid at 5 levels was used with parallel implementations, and the fully overlapped versions were used. Problem size scales with the number of GPUs, with 513^3 on 2 GPUs and 1025^3 using 16 GPUs on the NCSA Lincoln Tesla cluster. The smoother is a weighted Jacobi with $w = 0.86$ and 4 pre-smoothing and 4 post-smoothing iterations at each grid level. V-cycles and CUDA streams overlapping were used for each, and all computations were in single precision. The chart legend indicates the percentage of the multigrid solver time which was taken by that component.

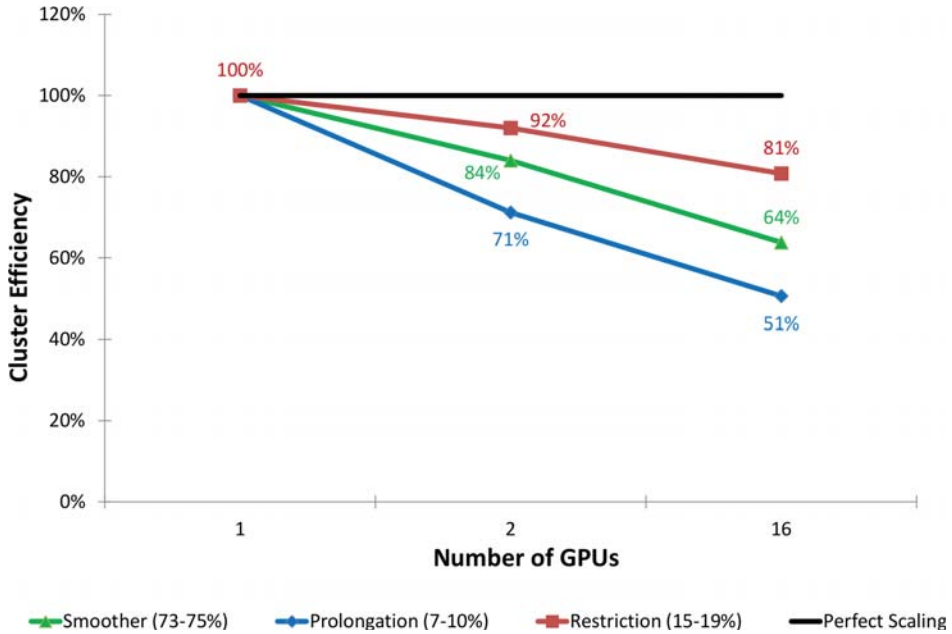


Figure 12. Weak scaling performance of the multigrid solver components using the TACC Longhorn system. Better scalability is seen with the faster memory transfer and network speeds.

VI. Conclusions and Future Work

We describe the implementation and performance of a dual-level (MPI-CUDA) parallel geometric multigrid solver used in an incompressible fluid flow solver. We adopt NVIDIA's CUDA programming model for fine-grain data-parallel operations within each GPU, and MPI for coarse-grain parallelization across the cluster. Numerical and performance results are shown to verify the solver's functionality. Our results show that this approach can improve the overall application performance in this application, and demonstrate how this approach can be adopted by similar applications.

A numerically efficient Poisson solver is crucial in CFD applications for conservation of physical quantities. For simulations that need high accuracy or time-accurate solutions, the addition of a parallel multigrid pressure Poisson solver allows rapid convergence of the pressure at each timestep even with very large models. Multigrid solvers introduce software complexity and require attention to detail to achieve the best convergence rates. Each part of the GPU-enabled multigrid solver has been examined, including convergence and performance of the Jacobi, Red-Black Gauss-Seidel, and Successive Overrelaxation smoothers on both the newest Fermi architecture and the previous generation. Early multigrid truncation with an approximate coarse grid solve was examined, and while some truncation is possible with little difference, more than a few of the coarsest levels skipped leads to slow convergence. As with the Jacobi solver, overlapping of the CUDA kernels with GPU data transfers and MPI network communication is investigated. With V-cycles, there is a distinct performance benefit to each level of overlapping, with the fully overlapped implementation almost 40% faster on 8 GPUs.

Coarse grid solvers are a important feature of parallel multigrid solvers, and numerous methods have been proposed for their solution. This paper presents a new method: embedded multigrid with amalgamation. In this technique, the coarsest grid is assembled on a single GPU where it is solved with a single-GPU multigrid implementation, with identical logic to the outer multigrid solution. This combines minimization of network communication on small grids, amalgamation to provide the GPU with enough work that it operates efficiently, and the excellent convergence rate of multigrid for the coarse grid.

Future work in this area includes investigation of domain decomposition methods as well as a deeper investigation of the amalgamation technique. While the 1D inter-GPU decomposition shows advantages on problems with growth in primarily 1D (channel flow) and 2D (lid-driven cavity and urban flow), the 3D growth examples in this study indicate it becomes inefficient for 32 or more nodes. The zero copy feature of CUDA 2.2 and later can be used to effectively

overlap the domain edge memory transfer along with the gather / scatter kernels. The amalgamation method used in this study has a sharp transition from using all available GPUs to only one, which could be improved. One possibility is stepwise merging where every other GPU merges with its neighbor, another is to choose discrete merge points such as $N \Rightarrow 16 \Rightarrow 4 \Rightarrow 1$ GPU. Another useful line of investigation would be looking at alternate solvers such as the conjugate gradient method for the coarse grid solver, in combination or contrast with the amalgamation method.

Acknowledgments

This work is partially funded by grants from NASA Idaho Space Grant Consortium and National Science Foundation (NSF) (Award #1043107). We utilized the Lincoln Tesla Cluster at the National Center for Supercomputing Applications and the Longhorn visualization cluster at the Texas Advanced Computing under grant number ASC090054 and ATM100032 from NSF TeraGrid. We thank NVIDIA Corporation for hardware donations and extend our thanks to Marty Lukes of Boise State University for his continuous help with building and maintaining our GPU computing infrastructure.

References

[1]Showerman, M., Enos, J., Pant, A., Kindratenko, V., Steffen, C., Pennington, R., and Hwu, W.-M., "QP: A Heterogeneous Multi-Accelerator Cluster," *Proceedings of the 10th LCD International Conference on High-Performance Clustered Computing*, Boulder, Colorado, March 10–12 2009.

[2]NCSA, "Intel 64 Tesla Linux Cluster Lincoln webpage," http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/, 2008.

[3]Navratil, P. A., "Introduction to Longhorn and GPU-Based Computing," *4th Iberian Grid Infrastructure Conference*, May 2010.

[4]Matsuoka, S., Aoki, T., Endo, T., Nukada, A., Kato, T., and Hasegawa, A., "GPU accelerated computingfrom hype to mainstream, the rebirth of vector computing," *Journal of Physics: Conference Series*, Vol. 180, No. 1, 2009, pp. 012043.

[5]Jacobsen, D. A., Thibault, J. C., and Senocak, I., "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," *48th AIAA Aerospace Science Meeting*, Jan. 2010.

[6]Jeong, B., Abram, G., Johnson, G., and Navrátil, P., "Large-Scale Visualization Using a GPU Cluster," *2nd NVIDIA GPU Technical Conference*, Sept. 2010.

[7]Phillips, J. C., Stone, J. E., and Schulten, K., "Adapting a Message-Driven Parallel Application to GPU-Accelerated Clusters," *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[8]Tokyo Institute of Technology, "TSUBAME2 System Architecture," http://tsubame.gsic.titech.ac.jp/en/tsubame2-system-architecture, Nov. 2010.

[9]HPCwire, "China's New Nebulae Supercomputer Is No. 2, Right on the Tail of ORNL's Jaguar," http://www.hpcwire.com/offthewire/Chinas-New-Nebulae-Supercomputer-Is-No-2-Right-on-the-Tail-of-ORNLs-Jaguar-95258669.html, May 2010.

[10]HPCwire, "NVIDIA Tesla GPUs Power World's Fastest Supercomputer," http://www.hpcwire.com/offthewire/NVIDIA-Tesla-GPUs-Power-Worlds-Fastest-Supercomputer-105983244.html, Oct. 2010.

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

[12]Kindratenko, V., Enos, J. J., Shi, G., Showerman, M. T., Arnold, G. W., Stone, J. E., Phillips, J. C., and mei Hwu, W., "GPU Clusters for High-Performance Computing," *Proceedings of the IEEE Workshop on Parallel Programming on Accelerator Clusters*, Aug. 2009.

[13]Brandvik, T. and Pullan, G., "Acceleration of a 3D Euler Solver using Commodity Graphics Hardware," *46th AIAA Aerospace Sciences Meeting and Exhibit*, Jan. 2008.

[14]Elsen, E., LeGresley, P., and Darve, E., "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Computational Physics*, Vol. 227, No. 24, Dec. 2008, pp. 10148–10161.

[15]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," *International Journal of Computational Science and Engineering (IJCSE)*, Vol. 4, No. 1, 2008, pp. 36–55.

[16]Cohen, J. M. and Molemaker, M. J., "A Fast Double Precision CFD Code using CUDA," *Proceedings of Parallel CFD*, 2009.

[17]Schive, H., Tsai, Y., and Chiueh, T., "GAMER: a GPU-Accelerated Adaptive Mesh Refinement Code for Astrophysics," *Astrophysical Journal Supplement Series*, Vol. 186, 2010, pp. 457–484.

[18]Thibault, J. C. and Senocak, I., "CUDA Implementation of a Navier–Stokes Solver on Multi-GPU Platforms for Incompressible Flows," *47th AIAA Aerospace Science Meeting*, Jan. 2009.

[19]Corrigan, A., Camelli, F. F., Löhner, R., and Wallin, J., "Running Unstructured grid-based CFD solvers on modern graphics hardware," *International Journal for Numerical Methods in Fluids*, Feb. 2010.

[20]Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical Recipes 3rd Edition: The Art of Scientific Computing*, Cambridge University Press, New York, NY, USA, 2007.

[21]Briggs, W. L., Henson, V. E., and McCormick, S. F., *A Multigrid Tutorial (2nd ed.)*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[22]Trottenberg, U., Oosterlee, C., and Schüller, A., *Multigrid*, Elsevier, 2001.

[23]McBryan, O. A., Frederickson, P. O., Linden, J., Schüller, A., Solchenbach, K., Stüben, K., Thole, C.-A., and Trottenberg, U., "Multigrid methods on parallel computers—a survey of recent developments," *IMPACT Comput. Sci. Eng.*, Vol. 3, No. 1, 1991, pp. 1–75.

[24]Chow, E., Falgout, R. D., Hu, J. J., Tuminaro, R. S., and Yang, U. M., *A Survey of Parallelization Techniques for Multigrid Solvers*, chap. 10, SIAM Series on Software, Environments, and Tools, SIAM, 2006.

[25]Göddeke, D., *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters*, Ph.D. thesis, Technischen Universität Dortmund, Dortmond, Germany, Feb. 2010.

[26]Goodnight, N., Lewin, G., Luebke, D., and Skadron, K., "A Multigrid Solver for Boundary-Value Problems Using Programmable Graphics Hardware," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, 2003, pp. 102–111.

[27]Kundu, P. K. and Cohen, I. M., *Fluid Mechanics*, Academic Press, 4th ed., 2007.

[28]Griebel, M., Dornseifer, T., and Neunhoeffer, T., *Numerical Simulation in Fluid Dynamics: A Practical Introduction*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[29]Tannehill, J. C., Anderson, D. A., and Pletcher, R. H., *Computational Fluid Mechanics and Heat Transfer*, Taylor & Francis, 2nd ed., 1997.

[30]Chorin, A. J., "Numerical Solution of the Navier–Stokes Equations," *Math. Comput.*, Vol. 22, 1968, pp. 745–762.

[31]Ghia, U., Ghia, K. N., and Shin, C., "High-Re Solutions for Incompressible Flow Using the Navier–Stokes Equations and a Multigrid Method," *Journal of Computational Physics*, Vol. 48, 1982, pp. 387–411.

[32]Wan, D. C., Patnaik, B. S. V., and Wei, G. W., "A New Benchmark Quality Solution for the Buoyancy-Driven Cavity by Discrete Singular Convolution," *Numerical Heat Transfer, Part B: Fundamentals*, Vol. 40, No. 3, 2001, pp. 199–228.

[33]Brandt, A., "Multi-level adaptive technique (MLAT) for fast numerical solutions to boundary value problems," *Lecture Notes in Physics 18*, edited by H. Cabannes and R. Temam, Proc. 3rd Int. Conf. Numerical Methods in Fluid Mechanics, Springer, Berlin, 1973, pp. 82–89.

[34]Brandt, A., "Multi-Level Adaptive Solutions to Boundary-Value Problems," *Math. Comput.*, Vol. 31, No. 138, April 1977, pp. 333–390.

[35]NVIDIA, *NVIDIA CUDA Programming Guide 3.1.1*, 2010.

[36]Hempel, R., "The MPI Standard for Message Passing," *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1994, Munich, Germany, April 18-20, 1994, Proceedings, Volume II: Networking and Tools*, edited by W. Gentzsch and U. Harms, Vol. 797 of *Lecture Notes in Computer Science*, Springer, 1994, pp. 247–252.

[37]Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S., "GPU Cluster for High Performance Computing," *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, 2004, pp. 47+.

[38]Micikevicius, P., "3D Finite Difference Computation on GPUs using CUDA," *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009, pp. 79–84.

[39]Gropp, W. D., "Parallel Computing and Domain Decomposition," *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, edited by T. F. Chan, D. E. Keyes, G. A. Meurant, J. S. Scroggs, and R. G. Voigt, SIAM, Philadelphia, PA, USA, 1992.