1-1-2010

An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters

Dana A. Jacobsen
Boise State University

Julien C. Thibault
Boise State University

Inanc Senocak
Boise State University

48th AIAA Aerospace Sciences Meeting <br>Including the <br>New Horizons Forum and Aerospace
4 - 7 January 2010, Orlando, Florida

AIAA 2010-522

An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters

Dana A. Jacobsen*, Julien C. Thibault*, and Inanc Senocak†

Boise State University, Boise, Idaho, 83725

Modern graphics processing units (GPUs) with many-core architectures have emerged as general-purpose parallel computing platforms that can accelerate simulation science applications tremendously. While multi-GPU workstations with several TeraFLOPS of peak computing power are available to accelerate computational problems, larger problems require even more resources. Conventional clusters of central processing units (CPU) are now being augmented with multiple GPUs in each compute-node to tackle large problems. The heterogeneous architecture of a multi-GPU cluster with a deep memory hierarchy creates unique challenges in developing scalable and efficient simulation codes. In this study, we pursue mixed MPI-CUDA implementations and investigate three strategies to probe the efficiency and scalability of incompressible flow computations on the Lincoln Tesla cluster at the National Center for Supercomputing Applications (NCSA). We exploit some of the advanced features of MPI and CUDA programming to overlap both GPU data transfer and MPI communications with computations on the GPU. We sustain approximately 2.4 TeraFLOPS on the 64 nodes of the NCSA Lincoln Tesla cluster using 128 GPUs with a total of 30,720 processing elements. Our results demonstrate that multi-GPU clusters can substantially accelerate computational fluid dynamics (CFD) simulations.

I. Introduction

Supercomputing is a dynamic field with continuous innovations in computing hardware and programming models. For instance, the use of clusters of commodity-based computers, so-called Beowulf clusters, started a shift in supercomputing about fifteen years ago[1]. Today this change is widely adopted in both industry and academia. With the cost effectiveness of clusters and the portable common Application Programming Interfaces (APIs) such as Message-Passing Interface (MPI)[2], the computational science and engineering (CSE) community has made huge strides towards a simulation-based predictive science capability in various domains.

Recent years have seen another shift in supercomputing with the introduction of general programming APIs to perform scientific computing on many-core GPUs[3]. The latest GPU programming interfaces such as NVIDIA's Compute Unified Device Architecture (CUDA)[4], and more recently Open Computing Language (OpenCL)[5] provide the programmer a flexible model while exposing enough of the hardware to allow the resources to be well utilized. Current high-end GPUs can achieve floating point throughputs that are orders of magnitude greater than that of modern CPUs by using a combination of highly parallel processing (200-800 processors per GPU), high memory bandwidth (typically a factor of 20 more than CPU to memory interfaces in commodity computers), and efficient thread scheduling methods. Equally important, latest motherboard designs can support as many as eight GPUs in a single compute node.[6] A multi-GPU cluster, where each compute-node of the cluster has at least two GPUs, can then be constructed by interconnecting the nodes with a fast network[7].

Multi-GPU computing is now a part of the supercomputing field. The National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana Champaign deployed the Lincoln Tesla Linux cluster for GPU computing in February 2009[8]. We expect wide adoption of multi-GPU clusters in industry and academia because both CPU and GPU based applications can be supported. This is a unique opportunity because simulation problems that were feasible only on leadership supercomputing facilities are now possible on more widely available systems. One of the challenges ahead of the scientific computing community is to develop and deploy scalable and efficient applications that can utilize the immense computational power of multi-GPU clusters. In the following, we investigate

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

several performance metrics using an incompressible flow solver[9], and make a case in favor GPU computing. We then extend our incompressible flow solver to multi-GPU clusters with a mixed MPI-CUDA implementation. We explore three programming strategies to successively boost the parallel performance on the NCSA Lincoln Tesla cluster.

II. GPU Hardware and Clusters

CUDA is the software and hardware architecture for GPUs produced by NVIDIA. The Tesla series of computing servers are distinguished by their large memory (1.5GB in the older C870 model, 4GB in the latest C1060 model, 6GB in the upcoming Tesla models with Fermi architecture) and lack of video output. The large memory size in particular makes the Tesla line of GPUs attractive for scientific computation. Tesla series have more rigorous testing which may result in less downtime and lost results. Tesla S870 and S1070 are composed of four GPUs and have a rack mount 1U form factor that makes installation in large clusters much easier. CUDA is also the software and hardware architecture for consumer models (e.g. GTX 200 series) that are sold for general graphics tasks at a reduced price. Consumer models can also be used for scientific computation. However, the amount of device memory (0.5 to 1GB on most models) on consumer models can be a limiting factor for large computational problems. Larger memory on each individual GPU reduces the total number of nodes needed for a given problem size, which means fewer or smaller network switches and reduced system cost. The price of fast networking (e.g. Infiniband) can be significant compared to compute hardware, hence is a major item in cost analysis of a GPU cluster.

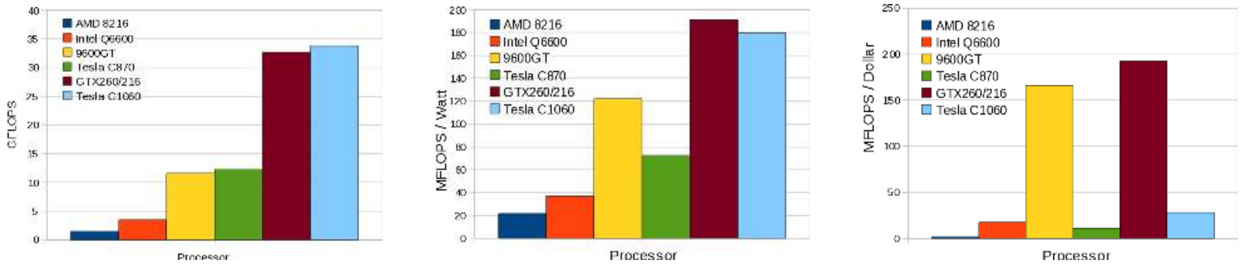
	AMD 8216	Intel Q6600	9600GT	Tesla C870	GTX260/216	Tesla C1060
Type	CPU	CPU	GPU	GPU	GPU	GPU
Date Introduced	Aug 2006	Jan 2007	Feb 2008	June 2007	Dec 2008	June 2008
Processing Units	2	4	64	216	216	240
Compute Capability	N/A	N/A	CC1.1	CC1.1	CC1.3	CC1.3
Device Memory	N/A	N/A	512MB	1536MB	896MB	4096MB
Power	68W	95W	95W	170W	171W	188W
Street Price	$800	$200	$70	$1,100	$170	$1,200
Peak GFLOPS	19.2	76.8	336	519	805	936
Sustained GFLOPS	1.47	3.52	11.6	12.3	32.7	33.8

Table 1. Six selected computing platforms. Street price in US dollars as of 1 December 2009. Sustained GFLOPS numbers are based on the performance of an incompressible flow lid-driven cavity simulation small enough to fit on all devices.

The cost and energy consumption of modern supercomputers (e.g. Roadrunner at Los Alamos National Laboratory and Jaguar at Oak Ridge National Laboratory) are substantial. In addition to the raw computational power, metrics such as price/performance and power/performance are valuable in assessing the potential of new computing hardware. Table 1 shows configuration data and single device computational performance of the CFD code used in the present study on a selection of contemporary CPU and GPU platforms. In the comparisons, we consider a lid-driven cavity benchmark problem[10] that is small enough ($256 \times 32 \times 256$) to entirely fit in the memory space of the smallest GPU. On the CPU side, dual-core AMD Opteron 8216 and quad-core Intel Core2 Duo Q6600 were considered using all the cores. On the GPU side, we compared the consumer models NVIDIA 9600GT and NVIDIA GTX 260 Core 216 and the high performance computing models NVIDIA Tesla C870 and C1060. We used Pthreads to exploit the multiple cores on the CPU devices and CUDA to utilize the multiple processors on the GPUs. All calculations were done using single precision floating point. The code is capable of supporting double precision on GPUs with compute capability 1.3 or higher.

The peak single-precision GFLOPS (billions of floating point operations per second) based on manufacturer specifications are also given in Table 1. NVIDIA GPUs are capable of one single-precision multiply-add and one single precision multiply per clock cycle per thread processor. The theoretical peak flops is therefore 3 flops × clock rate × number of thread processors. The Intel Core2 architecture is capable of 4 single precision multiply-adds per cycle per core using SSE4, therefore the Intel Q6600 can sustain 4 cores × 2.40 GHz × 8 flops = 76.8 single precision GFLOPS (Intel gives official numbers[11] for double precision). The AMD Opteron (K8) can use its 64-bit SSE instructions to sustain 4 single precision operations per cycle hence can sustain 2 cores × 2.40 GHz × 4 flops = 19.2 single precision

GFLOPS.



a) b) c)

Figure 1. Three performance metrics on six selected CPU and GPU devices based on incompressible flow computations on a single device. Actual sustained performance is used rather than peak device performance. a) Sustained GFLOPS, b) MFLOPS/Watt, c) MFLOPS/Dollar

Figure 1a shows three performance metrics on each platform using an incompressible flow CFD code. The GPU version of the CFD code is clearly an improvement over the Pthreads shared-memory parallel CPU version. Both of these implementations are written in C and use identical numerical methods[9]. The main impact on individual GPU performance was the introduction of compute capability 1.3, which greatly reduces the memory latency in some computing kernels due to the relaxed memory coalescing rules[4]. Compute capability 1.3 also added support for double precision which is important in many solutions.

Figure 1b shows the performance relative to the peak electrical power of the device. GPU devices show a definite advantage over the CPUs in terms of energy-efficient computing. The consumer video cards have a slight power advantage over the Tesla series, partly explained by having significantly less active global memory. The recent paper by Kindratenko et al.[12] details the measured power use of two clusters built using NVIDIA Tesla S1070 accelerators. They find significant power usage from intensive global memory accesses, implying CUDA kernels using shared memory not only can achieve higher performance but can use less power at the same time. Approximately 70% of the S1070's peak power is used while running their molecular dynamics program NAMD. Figure 1c shows the performance relative to the street price of the device which sheds light on the cost effectiveness of GPU computing. The consumer GPUs are better in this regard, ignoring other factors such as the additional memory present in the compute server GPUs.

The rationale for clusters of GPU hardware is identical to that for CPU clusters – larger problems can be solved and total performance increases. Figure 1c indicates that clusters of commodity hardware can offer compelling price/performance benefits. By spreading the models over a cluster with multiple GPUs in each node, memory size limitations can be overcome such that inexpensive GPUs become practical for solving large computational problems. Today's motherboards can accommodate up to 8 GPUs in a single node[6], enabling large-scale compute power in small to medium size clusters. However, the resulting heterogeneous architecture with a deep memory hierarchy creates challenges in developing scalable and efficient simulation applications. In the following sections, we focus on maximizing performance on a multi-GPU cluster through a series of mixed MPI-CUDA implementations.

III. Related Work

GPU computing has evolved from hardware rendering pipelines that were not amenable to non-rendering tasks, to the modern General Purpose Graphics Processing Unit (GPGPU) paradigm. Owens et al.[13] survey the early history as well as the state of GPGPU computing in 2007. Early work on GPU computing is extensive and used custom programming to reshape the problem in a way that could be processed by a rendering pipeline, often one without 32-bit floating point support. The advent of DirectX 9 hardware in 2003 with floating point support, combined with early work on high level language support such as BrookGPU, Cg, and Sh, led to a rapid expansion of the field.[14–19]

Brandvik and Pullan[20] show the implementation of 2D and 3D Euler solver on a single GPU, showing 29× speedup for the 2D solver and 16× speedup for the 3D solver. One unique feature of their paper is the implementation of the solvers in both BrookGPU and CUDA. Elsen et al.[21] show the conversion of a subset of an existing Navier-Stokes solver to arrive at a BrookGPU version of a 3D compressible Euler solver on a single GPU. Significant effort went into efficiently handling the non-uniform mesh as well as geometric multi-grid on an irregular mesh. Measured speedups of $15\times$ to $40\times$ on complex geometries were obtained from a NVIDIA 8800GTX compared to a single core of a 2.4GHz Intel Core 2 Duo E6600. Tölke and Krafczyk[22] describe a 3D Lattice Boltzmann model (D3Q13) in detail along with a CUDA implementation. Their single GPU implementation on an NVIDIA 8800 Ultra achieves a speedup of $100\times$ over an Intel Xeon (noting that the CPU calculation was done in double precision and with an implementation of a more detailed model, making the speedup value not directly comparable). Simek et al.[23] detail performance gains on a variety of single GPU platforms for atmospheric dispersion simulations, achieving speedups as high as $77\times$ compared to a CPU implementation. Cohen and Molemaker[24] describe the implementation and validation of an incompressible Navier-Stokes solver with Boussinesq approximation that supports double precision. The numerical approach is very similar to our approach with the exception of a multigrid solver rather than a Jacobi iterative solver. Since it is a single GPU implementation, domain decomposition and overlapping methods are not examined. Performance of their implementation on a Tesla C1060 was $8\times$ faster than comparable multithreaded code running on an 8-core Intel Xeon E5420 at 2.5GHz. Double precision was 46% to 66% slower than single precision, which is in line with results we have seen with our model running in double precision.

As mentioned earlier, modern motherboards can accommodate multiple GPUs in a single workstation with several TeraFLOPS of peak performance. Currently, GPU programming models do not address parallel implementations for multi-GPU platforms. Therefore, GPU programming models have to be interleaved with MPI, OpenMP or Pthreads. In the multi-GPU computing front, Thibault and Senocak[9] developed a single-node multi-GPU 3D incompressible Navier-Stokes solver with a Pthreads-CUDA implementation. The GPU kernels from their study forms the internals of the present cluster implementation. Thibault and Senocak demonstrated a speedup of $21\times$ for two Tesla C870 GPUs compared to a single core of an Intel Core 2 3.0 GHz processor, $53\times$ for two GPUs compared to an AMD Opteron 2.4 GHz processor, and $100\times$ for four GPUs compared to the same AMD Opteron processor. Four GPUs were able to sustain $3\times$ speedup compared to a single GPU on a large problem size. The multi-GPU implementation of Thibault and Senocak does not overlap computation with GPU data exchanges. Overlapping features are introduced in the present study.

Micikevicius[25] describes both single and multi GPU CUDA implementations of a 3D 8^{th}-order finite difference wave equation computation in detail. The wave equation code is composed of a single kernel with one stencil operation. MPI was used for process communication in multi-GPU computing. Micikevicius uses a two phase computation where the cells to be exchanged are computed first, then the inner cells are computed in parallel with asynchronous memory copy operations and MPI exchanges. With efficient overlapping of computations and copy operations the author achieves superlinear speedup on 4 GPUs running on two Infiniband connected nodes with two Tesla 10-series GPUs each, when using a large enough dataset.

Several early studies have demonstrated the potential of GPU clusters to tackle computationally large problems. We note some of these studies use graphics languages rather than CUDA. Fan et al.[26] investigated GPU cluster use for scientific computation in their 2004 paper. Fan et al. showed an urban dispersion simulation implemented as a Lattice Boltzmann model (LBM) run on a GPU cluster. Their cluster consisted of 32 nodes each with a single GPU and uses MPI for inter-node communication. The paper emphasizes the importance of minimizing communication costs, and the authors give excellent views of overlapping communication and computation time in their model. Göddeke et al.[27] in 2007 surveys cluster approaches for scientific computation and shows how GPU clusters from commodity components can be a practical solution. Phillips et al.[28] in 2008 also investigate GPU clusters, detailing many of the complications that arise that are unique to the system. Schive et al.[29] detail a 16 node, 32 GPU cluster running a parallel direct N-body simulation system, achieving 7.1 TeraFLOPS and over 90% parallel efficiency. Performance on GPU clusters running N-body simulations are better than those reported for fluid dynamics solvers which are implemented with several kernels, as noted by the authors in a more recent article[30] that describes an Euler solver using adaptive mesh refinement. Clearly, the sustained performance on GPU computing platforms depends on the application.

Göddeke et al.[31] explore course and fine grain parallelism in a finite element model for fluids or solid mechanics computations on a GPU cluster. Göddeke et al.[32] described the application of their approach to a large-scale solver toolkit. The Navier-Stokes simulations in particular exhibited limited performance due to memory bandwidth and latency issues. Optimizations were also found to be more complicated than simpler models such as the ones they previously considered. While the small cluster speedup of a single kernel is good, acceleration of the entire model is a modest factor of two only. Their model uses a nonuniform grid and multigrid solvers within a finite element framework for relatively low Reynolds numbers.

Phillips et al.[28] describe many of the challenges that arise when implementing scientific computations on a GPU cluster, including the host/device memory traffic and overlapping execution with computation. A performance visualization tool was used to verify overlapping of CPU, GPU, and communication on an Infiniband connected 64 GPU cluster. Scalability is noticeably worse for the GPU accelerated application than the CPU application as the impact of the GPU acceleration is quickly dominated by the communication time. However, the speedup is still notable. Phillips et al.[33] describe a 2D Euler Equation solver running on an 8 node cluster with 32 GPUs. The decomposition is 1D, but GPU kernels are used to gather/scatter from linear memory to non-contiguous memory on the device.

IV. 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[34]:

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

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[35,36]

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

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 projection algorithm[37] is then adopted to find a numerical solution to the Navier-Stokes equation for incompressible fluid flows. In the present implementation, the pressure Poisson equation is solved using a Jacobi iterative solver. For time-accurate simulations, a more efficient solver such as a multi-grid method needs to be adopted. 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. We use single precision for all results shown in this paper.

V. Multi-GPU Cluster Implementation of a 3D Incompressible Navier-Stokes Solver

Multiple programming APIs 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. For problems that are small enough to run on a single GPU, overhead time is minimized as no GPU/host communication is performed during the computation, and all optimizations are done within the GPU code. 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[4]. 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.

POSIX Threads[38] (Pthreads) and OpenMP[39] are two APIs used for running parallel code on a single machine using shared memory, such as widely available symmetric multiprocessor machines. These APIs both use a shared memory space model. Combined with CUDA, multiple GPUs on a single computer can perform computation, copy their neighboring cells to the host, synchronize with their neighbor threads, and copy the received boundary cells to the GPU for use in the next computational step.

The Message Passing Interface (MPI) API is widely used for parallel programming on clusters. MPI works on both shared and distributed memory machines. In general it will have some performance loss compared to the shared memory model used by threading APIs such as OpenMP and Pthreads, but in return it offers a highly portable solution to writing programs to work on a wide variety of machines and hardware topologies. Using MPI with one process mapped to each GPU is the most straightforward way to use a multi-GPU cluster.

A. Domain Decomposition in the MPI-CUDA Implementation

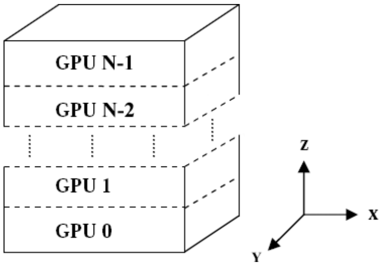


Figure 2. The decomposition of the full domain to the individual GPUs.

A 3D Cartesian volume is decomposed into 1D layers. These layers are then partitioned among the GPUs on the cluster to form a 1D domain decomposition. The 1D decomposition is shown in Fig. 2. After each GPU completes its computation the edge cells ("ghost cells") must be exchanged with neighboring GPUs. Efficiently performing this exchange process is crucial to cluster scalability.

While a 1D decomposition leads to more data being transferred as the number of GPUs increases, there are advantages to the method when using CUDA. In parallel CPU implementations, host memory access can be performed on non-contiguous segments with a relatively small performance loss. The `MPI_CART` routines supplied by MPI allow efficient management of virtual topologies, making the use of 2D and 3D decompositions easy and efficient. In contrast, the CUDA API only provides a way to transfer linear segments of memory between the host and the GPU. Hence, 2D or 3D decompositions for GPU implementations must either use nonstandard device memory layouts which result in poor GPU performance, or run separate kernels to perform gather/scatter operations into a linear buffer suitable for the `cudaMemcpy()` routine. These routines add significant time and hinder overlapping methods. For this reason, the 1D decomposition is deemed best for moderate size clusters such as the ones used in this study.

To accommodate overlapping, a further 1D decomposition is applied within each GPU. Figure 3 indicates how the 1D layers within each GPU are split into a top, bottom, and middle section. When overlapping communication and computation, the GPU executes each separately such that the memory transfers and MPI communication can happen simultaneously with the computation of the middle portion.

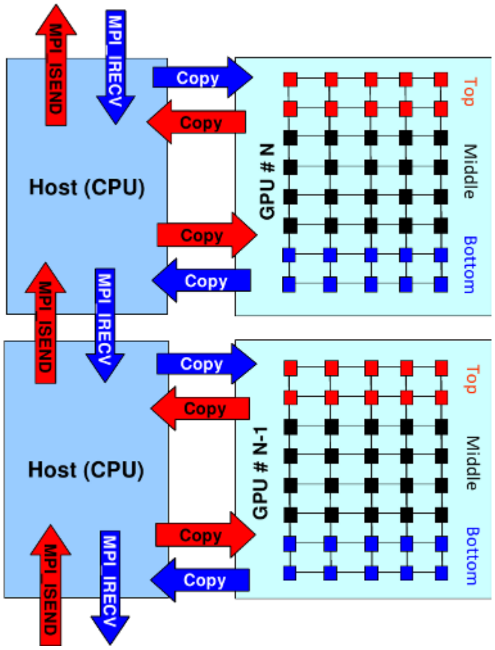


Figure 3. An overview of the MPI communication, GPU memory transfers, and the intra-GPU 1D decomposition used for overlapping.

B. MPI-CUDA Implementations

We developed three implementations of the incompressible flow solver. We successively overlap computations with inter-node and intra-node data exchanges to better utilize the cluster resources. All three of the implementations have much in common, with differences in the way data exchanges are implemented. We show in section VI.B. that implementation details in the data exchanges have a large impact on performance.

For all three implementations, one MPI process is started per GPU. Since we must ensure that each process is assigned a unique GPU identifier, an initial mapping of hosts to GPUs is performed. A master process gathers all the host names, assigns GPU identifiers to each host such that no process on the same host has the same identifier, and scatters the result back. At this point the `cudaSetDevice()` call is made on each process to map one of the GPUs to the process which assures that no other process on the same node will map to the same GPU.

The CUDA portions of the solver follow the same form shown in Thibault and Senocak[9]. These kernels implement the computation steps of the solver in the GPU, and do not require any modification for use in the multiple GPU implementation. We added the use of constant memory to support run-time model configuration while maintaining efficient GPU memory accesses to this common data. A temperature kernel was added and the momentum kernel changed to apply the buoyancy effect.

```
for (t=0; t < time_steps; t++)
{
    temperature <<<grid,block>>> (u,v,w,phiold,phi,phinew);
    ROTATE_POINTERS(phi,phinew);
    temperature_bc <<<grid,block>>> (phi);

    EXCHANGE(phi);

    momentum <<<grid,block>>> (phi,uold,u,unew,vold,v,vnew,wold,w,wnew);
    momentum_bc <<<grid,block>>> (unew,vnew,wnew);

    EXCHANGE(unew,vnew,wnew);

    divergence <<<grid,block>>>(unew,vnew,wnew,div);

    // Jacobi solver iteration
    for (m = 0; m < num_jacobi_iterations; m++)
    {
        pressure <<<grid,block>>> (div,p,pnew);
        ROTATE_POINTERS(p,pnew);
        pressure_bc <<<grid,block>>> (p);

        EXCHANGE(p);
    }
    correction <<<grid,block>>> (unew,vnew,wnew,p);
    momentum_bc <<<grid,block>>> (unew,vnew,wnew);

    EXCHANGE(unew,vnew,wnew);

    ROTATE_POINTERS(u,unew);
    ROTATE_POINTERS(v,vnew);
    ROTATE_POINTERS(w,wnew);
}
```

Listing 1. Host code for the projection algorithm to solve buoyancy driven incompressible flow equations on multi-GPU platforms. The outer loop is used for time stepping while the inner loop is in the iterative solution of the pressure Poisson equation. The `EXCHANGE` step updates the ghost cells for each GPU with the contents of the data from the neighboring GPU.

The projection algorithm is composed of distinct steps in the solution of the fluid flow equations. Listing 1 shows an outline of the basic implementation using CUDA kernels to perform each step. The steps marked as `EXCHANGE` are where ghost cells for each GPU are filled in with the calculated contents of their neighboring GPUs. The most basic exchange method is to call `cudaMemcpy()` to copy the edge data to host memory, MPI exchange using `MPI_Send` and `MPI_Recv`, and finally another `cudaMemcpy()` to copy the received edge data to device memory. This is

```
// PART 1:  Interleave non-blocking MPI calls with device
//          to host memory transfers of the edge layers.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north)

// ... other exchanges may be started here, before finishing in order

// PART 2:  Once MPI indicates the ghost layers have been received,
//          perform the host to device memory transfers.

MPI_Wait(receives for new ghost layers to complete)
cudaMemcpy(new north ghost layer from host to device)
cudaMemcpy(new south ghost layer from host to device)
```

Listing 2. An `EXCHANGE` operation overlaps GPU memory copy operations with asynchronous MPI calls for communication.

```
// The GPU domain is decomposed into three sections:
//    (1) top edge, (2) bottom edge, and (3) middle
// Which of them the kernel should process is indicated
// by a flag given as an argument.

pressure <<<grid_edge,block>>> (top_flag,    div,p,pnew);
pressure <<<grid_edge,block>>> (bottom_flag, div,p,pnew);

// The cudaMemcpy calls below will not start until
// the previous kernels have completed.
// This is identical to part 1 of the EXCHANGE operation.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north);

pressure <<<grid_middle,block>>> (middle_flag, div,p,pnew);

// This is identical to part 2 of the EXCHANGE operation.
MPI_Wait(receives for new ghost layers to complete)
cudaMemcpy(new north ghost layer from host to device)
cudaMemcpy(new south ghost layer from host to device)

pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Listing 3. The pressure loop where the CUDA kernel is split to overlap computation with MPI communication.

```
pressure <<<grid_edge,block, stream[0]>>> (top_flag,    div,p,pnew);
pressure <<<grid_edge,block, stream[1]>>> (bottom_flag, div,p,pnew);
// Ensure these kernels have finished before starting the copy
cudaThreadSynchronize();

cudaMemcpyAsync(south edge layer from device to host, stream[0])
cudaMemcpyAsync(north edge layer from device to host, stream[1])

pressure <<<grid_middle,block, stream[2]>>> (middle_flag, div,p,pnew);

MPI_Irecv(new ghost layer from north)
cudaStreamSynchronize(stream[0]);
MPI_Isend(south edge layer to south)

MPI_Irecv(new ghost layer from south)
cudaStreamSynchronize(stream[1]);
MPI_Isend(north edge layer to north);

MPI_Wait(south receive to complete)
cudaMemcpyAsync(new south ghost layer from host to device, stream[0])
MPI_Wait(north receive to complete)
cudaMemcpyAsync(new north ghost layer from host to device, stream[1])

// Ensure all streams are done, including copy operations and computation
cudaThreadSynchronize();
pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Listing 4. CUDA streams are used to fully overlap computation, memory copy operations, and MPI communication in the pressure loop.

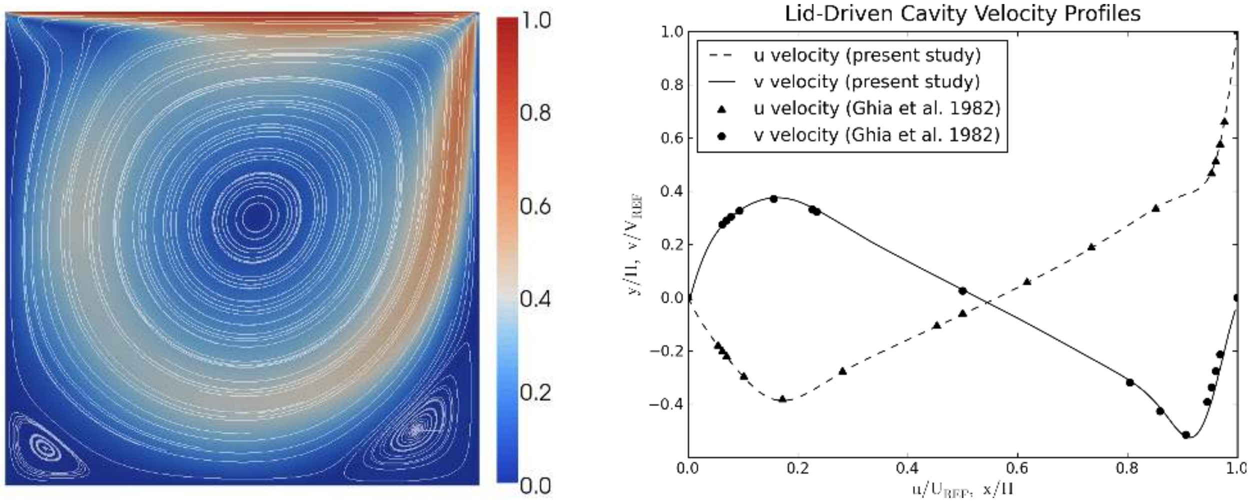
straightforward, but all calls are blocking which greatly hinders performance. Therefore we have not pursued this basic implementation.

1. Non-blocking MPI with No Overlapping of Computation

The first implementation we consider uses non-blocking MPI calls[40] to offer a substantial benefit over the blocking approach. It does not overlap computation although it tries to overlap memory copy operations. The basic EXCHANGE operation is shown in Listing 2. In this method, none of the device/host memory operations nor any MPI communication happens until the computation of the entire domain has completed. The MPI communication is able to overlap with the CUDA memory operations. When multiple arrays need to be exchanged, such as the three momentum components, the components may be interleaved such that the MPI send and receive for one edge of the first component is in progress while the memory copy operations for the later component are proceeding. This is done by starting part 1 for each component in succession, then part 2 for each component.

2. Overlapping Computation with MPI

The second implementation for exchanges aims to overlap the CUDA computation with the CUDA memory copy operations and the MPI communication. We split the the CUDA kernels into three calls such that the edges can be done separately from the middle. This has a very large impact on the cluster performance as long as the domain is large enough to give each GPU enough work to do. The body of the pressure kernel loop when using this method is shown in Listing 3. Rather than perform the computation on the entire domain before starting the exchange, the kernel is started with just the edges being computed. The first portion of the previously shown non-blocking MPI EXCHANGE operation is then started, which does device to host memory copy operations followed by non-blocking MPI communications. The computation on the middle portion of the domain can start as soon as the edge layers have finished transferring to the host, and operates in parallel with the MPI communication. The last part of the non-blocking MPI EXCHANGE operation is also identical and is run immediately after the middle computation is started. While this implementation results in significant overlap, it is possible to improve on it by overlapping the computation of the middle portion with the memory transfer of the edge layers as shown in the final implementation.



a)

b)

Figure 4. Lid-driven cavity simulation with Re=1000 on a $256 \times 32 \times 256$ 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.[10].

3. Overlapping Computation with MPI Communications and GPU Transfers

Our final implementation is enabled by CUDA streams, and uses asynchronous methods to start the computation of the middle portion as soon as possible, thereby overlapping computation, memory operations, and MPI communication. A similar approach is described in Micikevicius[25].This method has the highest amount over overlapping, and we expect it to have the best performance at large scales. The body of the pressure kernel loop when using this method is shown in Listing 4.

It is important to note that the computations inside the CUDA kernels need minimal change, and the same kernel can be used for all three implementations. A flag is sent to each kernel to indicate which portion (top, bottom, middle, or all) it is to compute, along with an adjustment of the CUDA grid size so the proper number of GPU threads are created. Since GPU kernels tend to be highly optimized, minimizing additional changes in kernel code is desirable.

VI. Results and Scaling

A. Numerical Results

We simulate the well-known lid-driven cavity[10] and natural convection in heated cavity[41] problems to validate the MPI-CUDA version of our CFD solver. Figure 4 presents the results of a lid-driven cavity simulation with a Reynolds number 1000 on a $256 \times 32 \times 256$ grid. Fig. 4a shows the velocity magnitude distribution and streamlines at mid-plane. As expected, the computations capture the two corner vortices at steady-state. In Fig. 4b, we compare horizontal and vertical components of the velocity along the centerlines to the benchmark data of Ghia et al.[10]. Our results agree well with the benchmark data.

We simulate the natural convection in a heated cavity problem to test our buoyancy-driven incompressible flow computations on a $128 \times 16 \times 128$ grid. A Prandtl number of 7 was used for each heated cavity simulation. Figure 5 presents the natural convection patterns and isotherms for Rayleigh numbers of 140 and 200,000. Lateral walls have constant temperature boundary conditions with one of the walls having a higher temperature than the wall on the opposite side. Top and bottom walls are insulated. Fluid inside the cavity is heated on the hot lateral wall and rises due to buoyancy effects, whereas on the cold wall it cools down and sinks, creating a circular convection pattern inside the cavity. Although not shown, our results agree well with similar results presented in Griebel et al.[35] Figure 6 presents a comparison of the horizontal centerline temperatures for a heated cavity with Ra=100,000 along with reference data from Wan et al.[41]. Our results are in good agreement.

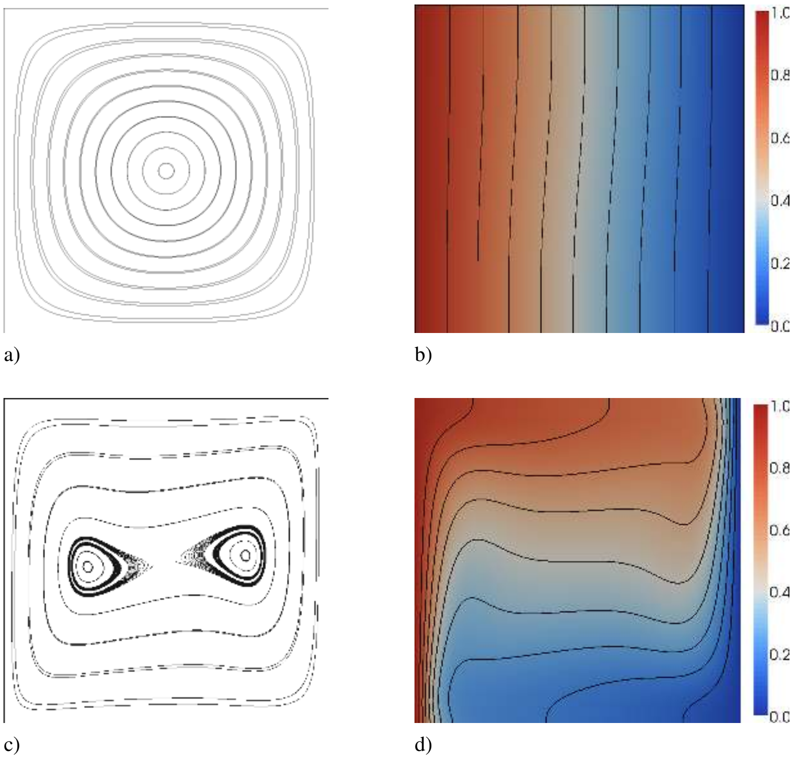


Figure 5. Natural convection in a cavity using a $128 \times 16 \times 128$ grid and Prandtl number 7, with a 2D center slice shown. a) Streamlines for Rayleigh number 140. b) Isotherms and temperature distribution for Rayleigh number 140. c) Streamlines for Rayleigh number 200,000. d) Isotherms and temperature distribution for Rayleigh number 200,000.

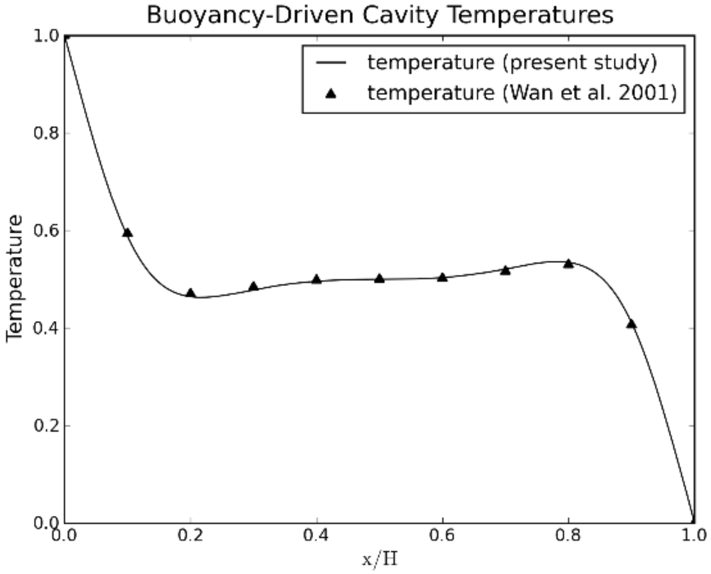


Figure 6. Centerline temperature for natural convection in a cavity with Prandtl number 7 and Rayleigh number 100,000, using a $256 \times 16 \times 256$ grid with a 2D center slice used. Comparison is shown to data from Wan et al.[41].

B. Performance Results from NCSA Lincoln Tesla Cluster

The NCSA Lincoln cluster consists of 192 Dell PowerEdge 1950 III servers connected via InfiniBand SDR (single data rate)[12]. Each compute node has two quad-core 2.33GHz Intel64 processors and 16GB of host memory. The cluster has 96 NVIDIA Tesla S1070 accelerator units each housing four C1060-equivalent Tesla GPUs. An accelerator unit is shared by two servers via PCI-Express ×8 connections. Hence, a compute-node has access to two GPUs. In this study we show performance measurements for 64 of the 192 available compute-nodes in the Lincoln Tesla cluster, with 128 GPUs being utilized.

Single GPU performance has been studied relative to a single CPU processor in various studies. Such performance comparisons are adequate for desktop GPU platforms. On a multi-GPU cluster, a fair comparison should be based on all the available CPU resources in the cluster. To partially address this issue, the CPU version of our CFD code is also parallelized with Pthreads to use the eight CPU cores available on a single compute-node of the Lincoln cluster. We note that CPU and GPU versions of our code are developed in a parallel effort using identical numerical methods[9].

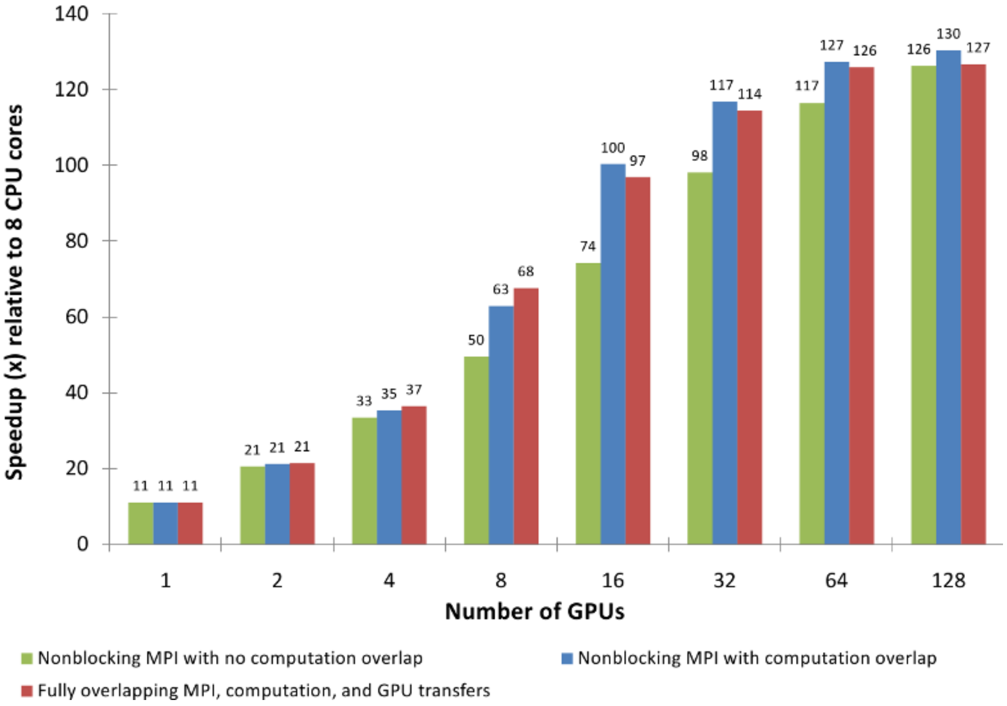


Figure 7. Speedup from the three MPI-CUDA implementations relative to the Pthreads parallel CPU code using all 8 cores on a compute-node. The lid-driven cavity problem is solved on a $1024 \times 64 \times 1024$ grid with fixed number of iterations and time steps.

We chose the lid-driven cavity problem at a Reynolds number of 1000 for performance measurements on the NCSA Lincoln Tesla cluster. Measurements were performed for both “strong scaling”, where the problem size remains fixed as the number of processing elements increases, and “weak scaling”, where the problem size grows in direct proportion to the number of processing elements. Measurements for the CPU application were done using a Pthreads shared-memory parallel implementation using all eight CPU cores on a single compute-node of the Lincoln cluster. All measurements include the complete time to run the application including setup and initialization, but do not include I/O time for writing out the results.

Figure 7 shows the speedup of the MPI-CUDA GPU application relative to the performance of the CPU application using Pthreads. The computational performance on a single compute-node with 2 GPUs was 21 times faster than 8 Intel Xeon cores, and 64 compute-nodes with 128 GPUs performed up to 130 times faster than 8 Intel Xeon cores. In all configurations the fully overlapped implementation performed faster than the first implementation that did not perform overlapping. Looking at the results for 16 to 128 GPUs, the second implementation which performs computation overlap shows a small advantage over the final fully overlapping implementation using CUDA streams. With the

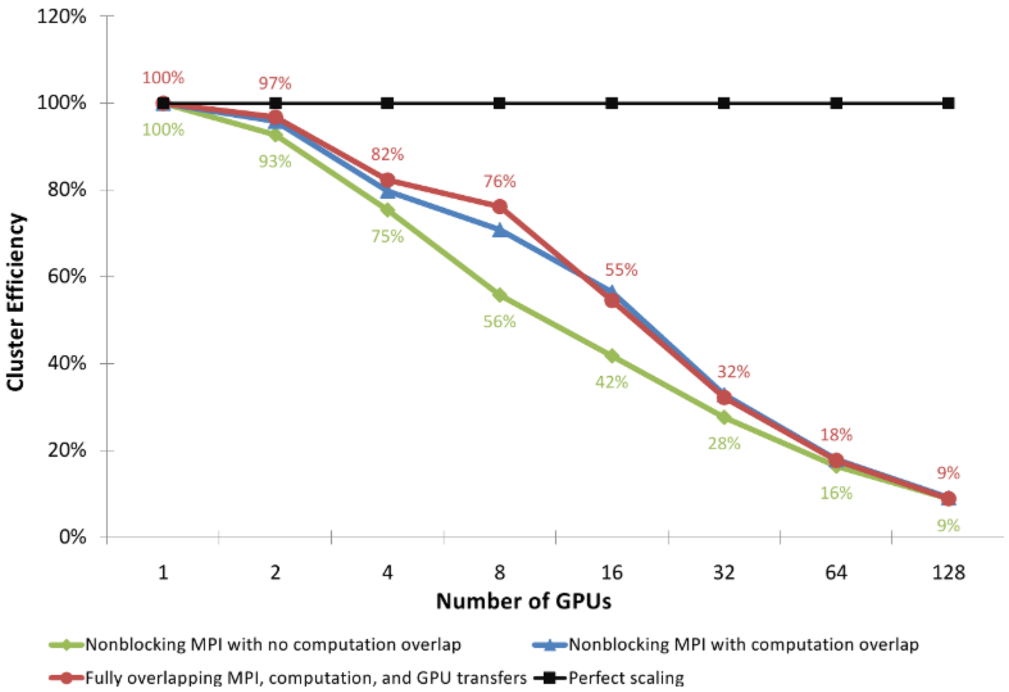


Figure 8. Efficiency of the three MPI-CUDA implementations with increasing number of GPUs (strong scalability presentation). The lid-driven cavity problem is solved on a $1024 \times 64 \times 1024$ grid with fixed number of iterations and time steps.

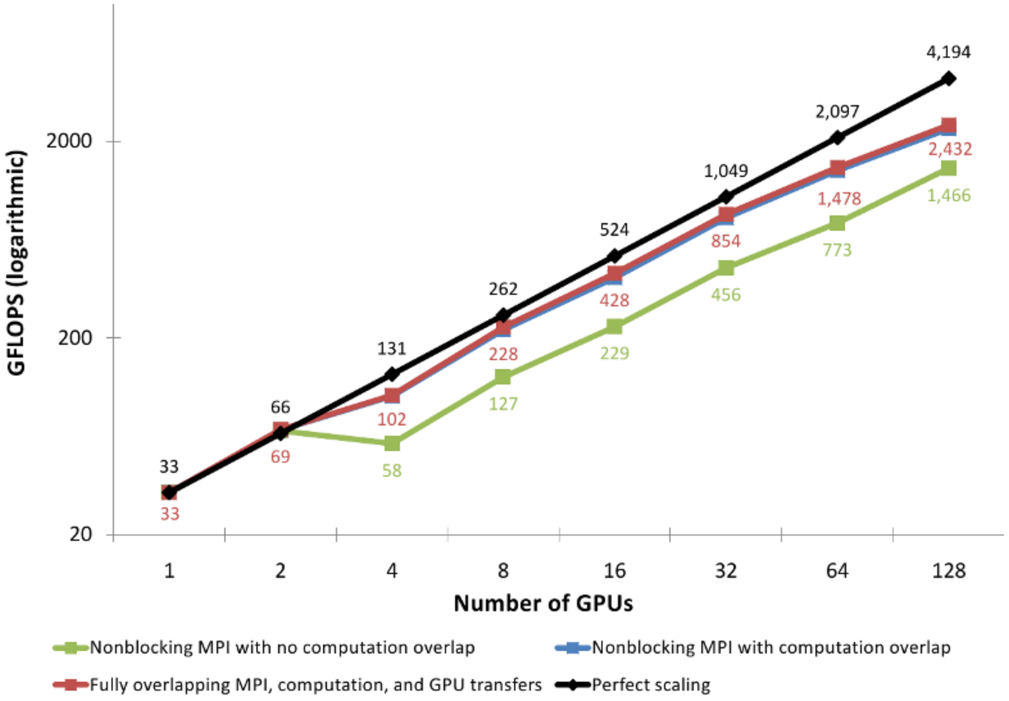


Figure 9. Sustained performance of the three MPI-CUDA implementations measured in GFLOPS. The size of the computational grid is varied from ($1024 \times 64 \times 1024$ to $11392 \times 64 \times 11776$) with increasing number of GPUs. Incompressible flow computations resulted in a sustained 2.4 TeraFLOPS performance with 128 GPUs on 64 compute-nodes.

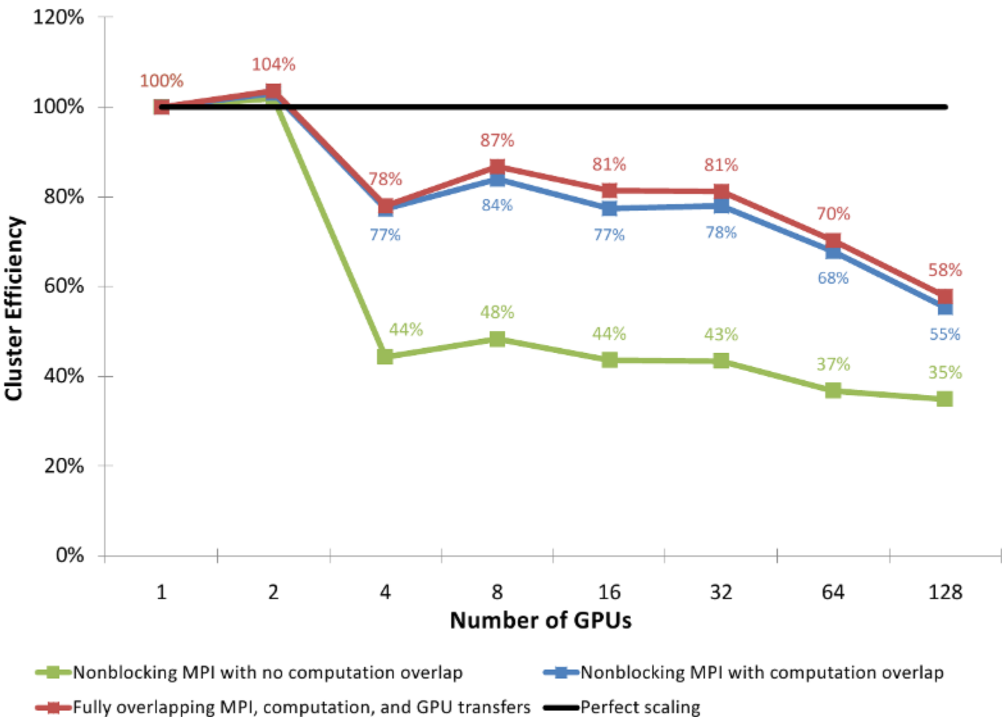


Figure 10. Efficiency of the three MPI-CUDA implementations with increasing number of GPUs (weak scalability presentation). The size of the computational grid is varied from $(1024 \times 64 \times 1024$ to $11392 \times 64 \times 11776)$ with increasing number of GPUs.

fixed problem size, the amount of work to do on each node quickly drops, with under 2ms of compute-time per pressure iteration. We think the additional overhead of CUDA streams creation and synchronization are leading to this difference.

The efficiency numbers for a fixed problem size in Figure 8 indicate the way scaling drops off as the number of nodes is increased. Linear speedup would result in 100% efficiency. Once network traffic becomes significant at 4 GPUs, the overlapping implementations show improved performance and efficiency. However, with the fixed size workload, eventually no amount of overlapping is able to overcome the network overhead once the amount of work per GPU becomes very small. With 128 GPUs, only 30MB of GPU memory is used per GPU for the fixed problem size considered. Although we have used 128 GPUs, there appears to be no significant gain in performance beyond 16 GPUs for our fixed-size problem which uses approximately 4GB of GPU memory.

All three MPI-CUDA implementations were also run with increasing problem sizes such that the memory used per GPU was approximately equal. This is commonly referred to as weak scalability. Figure 9 shows the sustained GFLOPS performance on a logarithmic scale. We deployed 128 GPUs on 64 compute-nodes to sustain a 2.4 TeraFLOPS performance with our fully overlapped implementation. With almost 500GB of memory used during the computation, it is not possible to directly compare this to a single node CPU implementation on traditional machines. Figure 10 shows much improved scaling compared to the fixed problem size case, and the advantage of overlapping computation and communication is also clear. Running with two GPUs results in slightly higher efficiency as the two GPUs are on a single compute-node resulting in no network latency, and the changed domain size within each GPU results in slightly faster performance. At four GPUs the internode network (Infiniband SDR) is active and the performance impact is immediate. As more GPUs are added, little additional efficiency is lost until 64 GPUs, where a downward trend is evident. Our results shed light into actual performance of multi-GPU clusters and could be helpful in future cluster designs.

We highlight a design issue with the NCSA Lincoln cluster that has an impact on our results. The connection between the compute-nodes and the Tesla GPUs are through PCI-Express Gen 2 ×8 connections rather than ×16. Measured bandwidth for pinned memory is approximately 1.6 GB/s, which is significantly slower than the 5.6 GB/s that we measured on a local workstation with PCIe Gen 2 ×16 connections to Tesla C1060s. On a multi-GPU cluster with a faster PCIe bus, performance for all three MPI-CUDA implementations are expected to improve. The fully overlapping method we show in Listing 4 would probably show the least benefit as it overlaps all device memory copies while computing.

VII. Conclusions

We have presented a dual-level parallel implementation of the Navier-Stokes equations to simulate buoyancy-driven incompressible fluid flows on multi-GPU clusters with heterogeneous architectures. 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. We have investigated the performance and scalability of incompressible flow computations on the NCSA Lincoln Tesla cluster with three different MPI-CUDA implementations. We adopt a 1D domain decomposition strategy as the overhead for gathering and scattering the data into linear transfer buffers can often exceed the advantages of the smaller transfer sizes that one could get from 2D or 3D domain decompositions. An additional level of 1D domain decomposition is also adopted within the compute-space of each GPU to overlap intra- and inter-node data exchanges with advanced features of MPI and CUDA.

Scaling and speedup results on the NCSA Lincoln Tesla cluster are very promising. Performance on a fixed problem size using 128 GPUs on 64 compute-nodes resulted in a speedup of 130× over the CPU solution using Pthreads on two quad-core 2.33GHz Intel Xeon processors. With a large problem size, we have sustained a performance of 2.4 TeraFLOPS using 128 GPUs on 64 compute-nodes of the NCSA Lincoln Tesla cluster. A total of 30,720 processing elements were deployed to process 8 billion grid cells. Our results demonstrate the cost and energy-efficiency of GPU computing. Multi-GPU clusters are powerful platforms to solve computationally large problems. With their heterogeneous architectures that can support both CPU and GPU based applications, we expect a wide adoption of multi-GPU clusters in the industry and academia.

Acknowledgments

We thank Prof. Wen-Mei Hwu of University of Illinois, Sumip Gupta and David Luebke of NVIDIA Corporation for their help with the resources. We extend our thanks to Marty Lukes of Boise State University for his continuous help with building and maintaining our GPU computing infrastructure, and to Timothy J. Barth of NASA Ames Research Center for many helpful discussions. This work is partially funded by research initiation grants from NASA-Idaho EPSCoR and the NASA Idaho Space Grant Consortium. The computations. We utilized the Lincoln Tesla Cluster at the National Center for Supercomputing Applications under grant number ASC090054.

References

[1] Sterling, T., Becker, D. J., Savarese, D., Dorband, J. E., Ranawake, U. A., and Packer, C. V., "Beowulf: A Parallel Workstation For Scientific Computation," *In Proceedings of the 24th International Conference on Parallel Processing*, CRC Press, 1995, pp. 11–14.

[2] 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.

[3] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Transactions on Graphics*, Vol. 23, ACM Press, New York, NY, USA, 2004, pp. 777–786.

[4] NVIDIA, *NVIDIA CUDA Programming Guide 2.0*, 2008.

[5] Khronos Group, *The OpenCL Specification: Version 1.0*, 2009.

[6] HPCwire, "Microway 9U Compact GPU Cluster with OctoPuter," Nov. 2009, http://www.microway.com/tesla/clusters.html.

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

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

[9] 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*, 2009.

[10] Ghia, U., Ghia, K., 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.

[11] Intel, "Intel microprocessor export compliance metrics," May 2009, http://www.intel.com/support/processors/sb/cs-023143.htm.

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

[13] 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.

[14] Bolz, J., Farmer, I., Grinspun, E., and Schröder, P., "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM SIGGRAPH 2003 Papers*, ACM, San Diego, California, 2003, pp. 917–924.

[15] 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.

[16] Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A., "Simulation of Cloud Dynamics on Graphics Hardware," *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, 2003, pp. 92–101.

[17] Krüger, J. and Westermann, R., "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," *ACM SIGGRAPH 2003 Papers*, ACM, 2003, pp. 908–916.

[18] Liu, Y., Liu, X., and Wu, E., "Real-Time 3D Fluid Simulation on GPU with Complex Obstacles," *Pacific Graphics 2004*, IEEE Computer Society, 2004, pp. 247–256.

[19] Zhao, Y., Qiu, F., Fan, Z., and Kaufman, A., "Flow Simulation with Locally-Refined LBM," *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, ACM, Seattle, Washington, 2007, pp. 181–188.

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

[21] 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.

[22] Tölke, J. and Krafczyk, M., "TeraFLOP computing on a desktop PC with GPUs for 3D CFD," *International Journal of Computational Fluid Dynamics*, Vol. 22, No. 7, 2008, pp. 443–456.

[23] Simek, V., Dvorak, R., Zboril, F., and Kunovsky, J., "Towards Accelerated Computation of Atmospheric Equations Using CUDA," *UKSim 2009: 11th International Conference on Computer Modelling and Simulation*, 2009, pp. 449–454.

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

[25] 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.

[26] 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+.

[27] 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 Computing, Special issue: High-performance computing using accelerators*, Vol. 33, No. 10–11, Nov. 2007, pp. 685–699.

[28] 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.

[29] Schive, H., Chien, C., Wong, S., Tsai, Y., and Chiueh, T., "Graphic-Card Cluster for Astrophysics (GraCCA) – Performance Tests," July 2007.

[30] Schive, H., Tsai, Y., and Chiueh, T., "GAMER: a GPU-Accelerated Adaptive Mesh Refinement Code for Astrophysics," July 2009.

[31] 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.

[32] Göddeke, D., Buijssen, S., Wobker, H., and Turek, S., "GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver," *International Conference on High Performance Computing & Simulation*, 2009, pp. 12–21.

[33] 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*, 2009.

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

[35] 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.

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

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

[38] Butenhof, D. R., *Programming with POSIX Threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[39] The OpenMP ARB, *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, Oct. 1997.

[40] Gropp, W., Thakur, R., and Lusk, E., *Using MPI-2: Advanced Features of the Message Passing Interface*, MIT Press, 1999.

[41] 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.