1-1-2009

CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows

Julien C. Thibault
Boise State University

Inanc Senocak
Boise State University

47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition
5 - 8 January 2009, Orlando, Florida

AIAA 2009-758

CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows

Julien C. Thibault[1] and Inanc Senocak[2]
Boise State University, Boise, Idaho, 83725

Graphics processor units (GPU) that are traditionally designed for graphics rendering have emerged as massively-parallel "co-processors" to the central processing unit (CPU). Small-footprint desktop supercomputers with hundreds of cores that can deliver teraflops peak performance at the price of conventional workstations have been realized. A computational fluid dynamics (CFD) simulation capability with rapid computational turn-around time has the potential to transform engineering analysis and design optimization procedures. We describe the implementation of a Navier-Stokes solver for incompressible fluid flow using desktop platforms equipped with multi-GPUs. Specifically, NVIDIA's Compute Unified Device Architecture (CUDA) programming model is used to implement the discretized form of the governing equations. The projection algorithm to solve the incompressible fluid flow equations is divided into distinct CUDA kernels, and a unique implementation that exploits the memory hierarchy of the CUDA programming model is suggested. Using a quad-GPU platform, we observe two orders of magnitude speedup relative to a serial CPU implementation. Our results demonstrate that multi-GPU desktops can serve as a cost-effective small-footprint parallel computing platform to accelerate CFD simulations substantially.

I. Introduction

In the last decade, CPU designers have focused on developing multi-core architectures instead of increasing the clock frequency by putting more transistors on the die because of power constraints[1]. GPU designers have adopted the many-core strategy early on, since graphics rendering is a parallel task. GPUs are based on the stream processing architecture[2] that is suitable for compute-intensive parallel tasks[3]. Modern GPUs can provide memory bandwidth and floating-point performances that are orders of magnitude faster than a standard CPU. Figure 1 depicts the growing gap in peak performance, measured in floating point operations per second (FLOPS) between GPU and CPU over the last five years. Currently, NVIDIA GPUs outperform Intel CPUs on floating point performance (Fig. 1) and memory bandwidth, both by a factor of roughly ten[3].

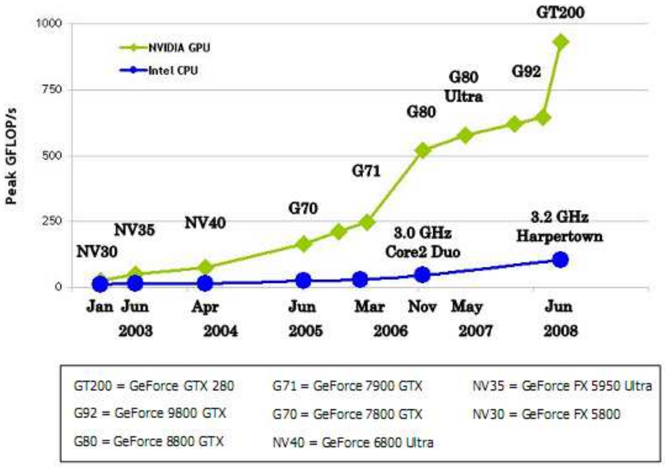

Figure 1. Evolution of floating-point performance for Intel CPUs and NVIDIA GPUs (courtesy of NVIDIA).

Until recently, using the GPUs for general-purpose computation was a complicated exercise. A good knowledge of graphics programming was required, because GPU's old fixed-function pipeline did not allow complex operations[4]. GPUs have evolved into a programmable engine, supported by new programming models trying to find the right balance between low access to the hardware and high-level programmability[4]. Brook programming model, released in 2004 by

[1] Graduate Research Assistant, Department of Computer Science, Student Member AIAA.
[2] Assistant Professor, Department of Mechanical & Biomedical Engineering, Member AIAA.

Stanford University, offered one of the first development platforms for general purpose GPU (GPGPU) programming[3,5]. Brook provides a GPU abstraction layer that enables data parallelism. It keeps the programmer away from having an extensive knowledge of graphics programming - like OpenGL - while being platform independent. NVIDIA recently released a more advanced programming model for its own line of GPUs: Compute Unified Device Architecture (CUDA)[3]. With CUDA, NVIDIA offers a common architecture and programming model for its own line of GPUs. The C-based application programming interface (API) of CUDA enables data-parallelism through the use of shared memory, but also computation parallelism thanks to the introduction of the thread and grid concepts. The CUDA programming model has found success in the GPGPU community. There is also a recent effort called MCUDA[6] to program multi-core CPU architectures. Advanced Micro Devices (AMD), on the other hand, offers Brook+, a modified version of the Brook open source compiler. Additionally, AMD's Compute Abstraction Layer (CAL) is used as a cross-platform interface to the GPU. Both AMD CAL and Brook+ are available in AMD's software development kit (SDK)[7].

Advances in many-core architectures have been tremendous, but using the full potential of many-core architectures is not an easy task. Engineers and scientists may need to rewrite and optimize their legacy sequential codes to harness the compute-power of modern day multi-core CPUs, and many-core GPUs. Message Passing Interface (MPI) programming[8] has been widely adopted in parallel scientific computations. MPI can be adopted for parallel computations both on shared and distributed memory systems, but it has better scaling properties for distributed memory systems by design[8]. MPI provides a high level API that allows programmers to transparently make use of multiple processors on both shared and distributed systems. The programmer does not have to deal with the details of the communication protocol between the nodes. On shared memory systems, Posix multithreading offers low level functions to implement multi-threaded systems, while OpenMP provides a certain abstraction layer[9], which makes it more accessible to software developers. In contrast, CUDA offers a different approach that specifically targets the many-cores on a single GPU. It is the programmer's responsibility to optimize the usage of the memory and the threads available on the streaming cores[10]. Implementation for multiple GPUs is explicitly performed by programmers, and multi-GPU parallelism is not currently addressed by CUDA.

Prior to the introduction of the CUDA and Brook programming models, several Navier-Stokes solvers have been implemented for the GPU. Harris[11] implemented a 3D solver to create a physically-based cloud simulation using the Cg programming language from NVIDIA. It is a high-level programming language for graphics on GPUs, which operates as a layer above OpenGL. His implementation was based on the "stable fluids" method proposed by Stam[12]. This method is adapted to graphics application because of the real-time visualization constraint. In Ref. 13 the Navier-Stokes equations are solved for flow around complex geometries following the work of Harris[11]. Due to its relative potential for easy parallelization, the Lattice-Boltzman method (LBM) has also been implemented in different studies addressing complex geometries. In Ref. 14, GPU implementation of LBM resulted in speedup of 15× relative to the CPU implementation. In Ref. 15, an LBM was implemented on a GPU cluster to calculate winds and contaminant dispersion in urban areas. A speedup of 4.6× relative to a CPU cluster was achieved in their study[15], which demonstrates that GPU clusters can serve as an efficient platform for scientific computing.

High performance parallel computing with CUDA has already attracted various scientists in several disciplines, such as molecular dynamics[16-18], computational biology[19], linear algebra[20,21], weather forecasting[22] and artificial intelligence[23]. In the computational fluid dynamics (CFD) field, Tolke and Krafczyk [24] implemented a 3D Lattice-Boltzman method for flow through a generic porous medium. They obtained a gain of up to two orders of magnitude with respect to the computational of an Intel Xeon 3.4GHz. Brandvik and Pullan[25] mapped 2D and 3D Euler solvers to the GPU using BrookGPU and CUDA programming models. For the CUDA version of the 3D Euler solver, their computations on NVIDIA 8800GTX showed a speedup of 16× over the CPU, whereas the BrookGPU implementation of the 3D Euler solver showed a modest speedup of only 3× on the ATI 1950XT. Molemaker et al.[26] developed a multi-grid method to solve the pressure Poisson equation. The CUDA implementation of the multi-grid pressure Poisson solver produced a speedup of 55× relative to a 2.2MHz AMD Opteron processor[26].

The recent literature attests to the compute-potential of GPU computing with new programming models. Numerous studies have adopted the CUDA programming model to numerical problems that have practical applications in engineering and science at large[27]. In this study, we present the implementation of a 3-D Cartesian-grid CFD code on multi-GPU/multi-CPU desktop platforms for incompressible fluid flow simulations. Specifically, we adopt the NVIDIA CUDA programming model to implement the discretized form of the Navier-Stokes equations on desktop platforms with multiple GPUs. Communication among GPUs is enabled with POSIX threading. We validate our multi-GPU parallel CFD code against the well established lid-driven cavity flow problem[28]. Several performance tests that assess the computational speedup of multi-GPU platforms relative to a serial CPU code are presented. To the best of our knowledge, our work is the first implementation of an incompressible flow Navier-Stokes solver on multi-GPU desktop platforms.

II. Governing Equations and Numerical Approach

A. Governing Equations of Incompressible Fluid Flows

The Navier-Stokes equations for incompressible fluid flows can be written as follows

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

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

where **u** is the velocity vector, *P* is the pressure, *ρ* is the density and *ν* is the kinematic viscosity.

B. Numerical Approach

Second-order accurate central difference scheme is used to discretize the advection and diffusion terms of the Navier-Stokes equations on a uniform staggered grid[29]. First-order accurate, explicit Euler scheme is used for the time derivative term. The projection algorithm[30] is then adopted to find a numerical solution to the Navier-Stokes equation for incompressible fluid flows. In the projection algorithm, the velocity field **u*** is predicted using the momentum equations without the pressure gradient term as follows[29, 30]

$$\mathbf{u}^* = \mathbf{u}^t + \Delta t(-\mathbf{u}^t \nabla \cdot \mathbf{u}^t + \nu \nabla^2 \mathbf{u}^t) \,, \tag{3}$$

where the index *t* and *Δt* represents the time level and time step size, respectively.

The predicted velocity field **u*** does not satisfy the divergence free condition because the pressure gradient term is not included in Eq. (3). By enforcing the divergence free condition on the velocity field at time (t+1), the following pressure Poisson equation can be derived from the momentum equations given in Eq. (2)

$$\nabla^2 P^{t+1} = \frac{\rho}{\Delta t}\nabla \cdot \mathbf{u}^* \;. \tag{4}$$

In the present study, the above equation is solved using a Jacobi iterative solver to time march the equations to a steady-state solution. For time-accurate simulations, a more efficient solver (e.g., geometric multi-grid method) should be adopted for time-accurate unsteady simulations. The pressure field at time (t+1) is then used to correct the predicted velocity field **u*** as follows

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

III. Programming Model for GPU Computing

CUDA is a new programming model developed by NVIDIA to harness the computational power of their GPUs. CUDA is an extension to the C programming language and it enables the developers to launch and manage massively parallel computations on the GPU. The reader is referred to the CUDA programming guide for more details[3]. In this section we summarize the GPU hardware and the programming model. Hereinafter, we interchangeably use the term “host” to refer to the CPU and the term “device” to refer to the GPU.

A. Hardware Architecture

GPUs are originally developed for graphics rendering that requires parallel computation with intense arithmetic operations. A GPU is a set of single instruction, multiple data (SIMD) multiprocessors. In GPU designs, transistors are devoted to data processing rather than data caching and flow control[4]. In the CUDA programming model, compute-intensive tasks of an application are grouped into an instruction set and passed on to the GPU such that each thread core works on different data but executes the same instruction[3]. The memory hierarchy of CUDA is similar to the memory hierarchy of a conventional multiprocessor. Closer to the core, the local registers allow fast ALU operations (L1 cache). The shared memory, seen by all the cores of a single multiprocessor, can be compared to a second-level cache (L2), as it provides a memory closer to the processors that will be used to store data that tend to be used over time by any core[3]. The difference in CUDA is that the programmer is responsible for the management of the shared memory or the "GPU cache". The last level in this hierarchy is the global memory. It can be accessed by any processor of the GPU, but for a higher latency cost. Threads can actually perform simultaneous scatter or simultaneous gather operations if those addresses are aligned in memory[3]. Coalesced memory access is crucial for superior kernel performance as it hides the latency of the global memory. The challenge for a CUDA software developer is then, not only the parallelization of the code, but also the optimization of the memory accesses by making the best use of the shared memory and the coalesced access to the global device memory .

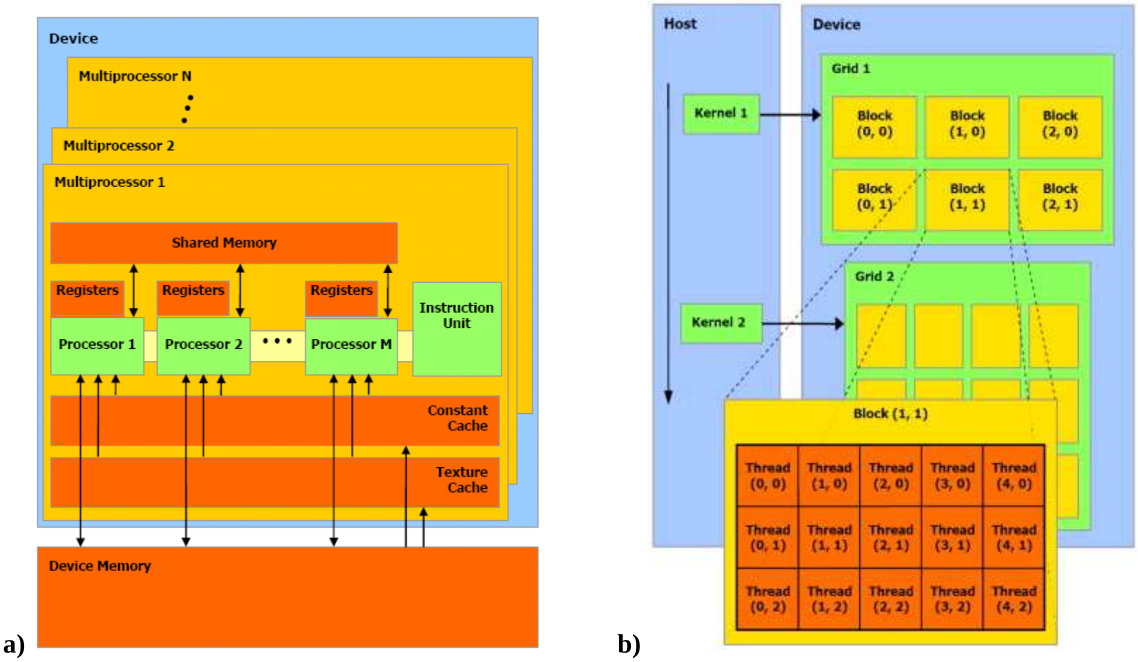

Figure 2. a) CUDA SIMD multiprocessor architecture (courtesy of NVIDIA). b) CUDA thread organization (courtesy of NVIDIA). In this example, the grid is composed of 3×2 blocks, each containing 5×3 threads.

Each multiprocessor also has read-only constant cache and texture cache. The constant cache can be used by the threads of a multiprocessor when trying to read the same constant value at the same time. Texture cache on the other hand is optimized for 2D spatial locality and should be preferred over global device memory when coalesced read cannot be achieved[3].

B. Programming model

The computation core of the CUDA programming model is the kernel, which is passed on to the GPU and executed by all the processor units, using different data streams. Figure 2b presents the layout of the threads in the CUDA programming model. Each kernel is launched from the host side (CPU), and it is mapped to a thread grid on the GPU. Each grid is composed of thread blocks. All the threads from a particular block have access to the same shared memory and can synchronize together. On the other hand, threads from different blocks cannot synchronize and can exchange data only through the global device memory[3]. A single block can only contain a limited number of threads, depending on the device model. But different blocks can be executed in parallel. Blocks executing the same kernel are batched together into a grid. The programmer needs to define the number of threads per block and the grid size (number of blocks) before launching the kernel. The parallel execution of the blocks is then handled by CUDA in a batch mode[3].

As mentioned earlier, CUDA API is an extension to the C programming language. It provides functions to manage the computations on the GPU. The full list of functions is discussed in detail in the CUDA programming guide[3]. Major functions that we have benefited in our study are `cudaMalloc()` and `cudaMemcpy()` functions. These functions allocate memory on the GPU and copy data from the CPU memory into the device memory of the GPU, respectively. `cudaFree()` function is used to free memory on the device. The kernel is launched by specifying the size of the grid (number of blocks) and the size of the blocks (number of threads) using the following prototype: `kernel_name<<grid size, block size>>()`. `__synchthreads()` can be used inside a kernel to synchronize all the threads of a same block. Global synchronization is not addressed by the CUDA model. The only way to force a global synchronization is to exit the kernel before launching a new one.

In addition, the CUDA API introduces the qualifiers `_shared_`, `_device_` and _constant_ to define the type of memory a variable should use. The function qualifiers `_device_`, `_global_`, and `_host_` specify whether the GPU or the CPU should execute and call the qualified function[3].

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

A. Single GPU Implementation

Let NX, NY and NZ be the number of computational nodes in the x, y and z directions for a flow domain, respectively. The 3D domain of size NX× NY×NZ is represented by a 2D matrix of width NX and height NY×NZ on the host side, as shown in Fig. 3. On the GPU side, the same representation is used to store data in global memory. This 2D mapping translates to efficient data transfer between the host (CPU) and the device (GPU). Note that several matrices are needed to represent the pressure and velocity components at different time levels. Memory allocation on the device is done only once before starting the time stepping.

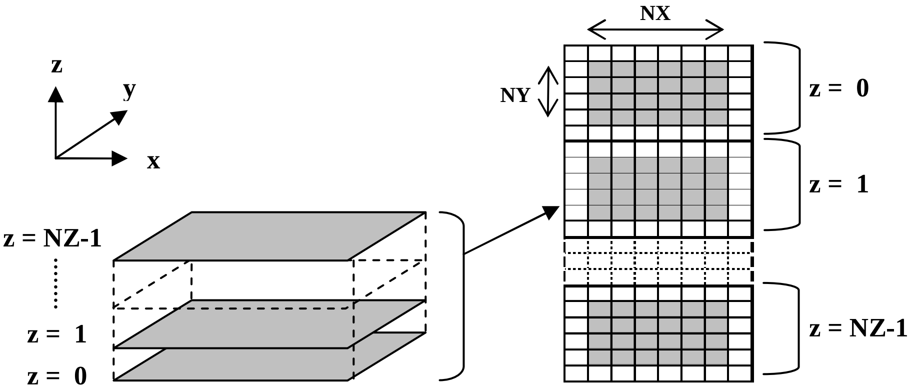

Figure 3. Mapping of a 3D computational domain to a 2D matrix. The mapping is used on both the CPU and the GPU sides. Cells in white on the 2D matrix represent the ghost (halo) cells to apply the boundary conditions.

Listing 1 shows the host side code for the time stepping. The code snippet is composed of two nested loops. The outer loop is used to advance the solution in time, and the inner loop is used for the iterations of the Jacobi solver to numerically solve the pressure Poisson equation (Eq. 6). In our implementation, the velocity field at time *t* depends only on the velocity field at *t-1*. Six different matrices are used to represent the velocity fields at the time *t* (`u, v, w`) and *t-1* (`uold, vold, wold`). The matrices are swapped at the end of each time step for reuse as shown in Listing 1. In a similar way, the Jacobi solver requires two matrices `p` and `pold`, which are swapped after each iteration of the Jacobi solver. As shown in Listing 1 the GPU code is composed of six different kernels to implement the major steps of the projection algorithm[30]. Separate kernels are needed to achieve global synchronization across the CUDA blocks before proceeding to the next time step for the computations.

```
//for each time step
for (t=0; t < ntstep; t++)
{
  //call kernel to compute momentum (ut, vt, wt)
  momentum << grid, block >> (u, v, w, uold, vold, wold)
  //call kernel to compute boundary conditions
  momentum_bc << grid, block >> (u, v, w)
  //call kernel to compute the divergence (div)
  divergence << grid, block >> (u, v, w, div)

  //for each Jacobi solver iteration
  for (j=0; j < njacobi; j++)
  {
    //call kernel to compute pressure
    pressure << grid, block >> (u, v, w, p, pold, div)
    //rotate matrices
    ptemp = pold; pold=p; p=ptemp;
    //call kernel to compute boundary conditions
    pressure_bc << grid, block >> (p)
  }
  //call kernel to correct velocity (ut, vt, wt)
  correction << grid, block >> (u, v, w, p)
  //call kernel to compute boundary conditions
  momentum_bc << grid, block >> (u, v, w)

  //rotate matrices
  utemp = uold; uold=u; u=utemp;
  vtemp = vold; vold=v; v=vtemp;
  wtemp = wold; wold=w; w=wtemp;
}
```

Listing 1. Partial host-side code that implements the projection algorithm[30] to solve the Navier-Stokes equations for incompressible fluid flow. The outer loop is used for time stepping while the inner loop is in the iterative solution of the pressure Poisson equation.

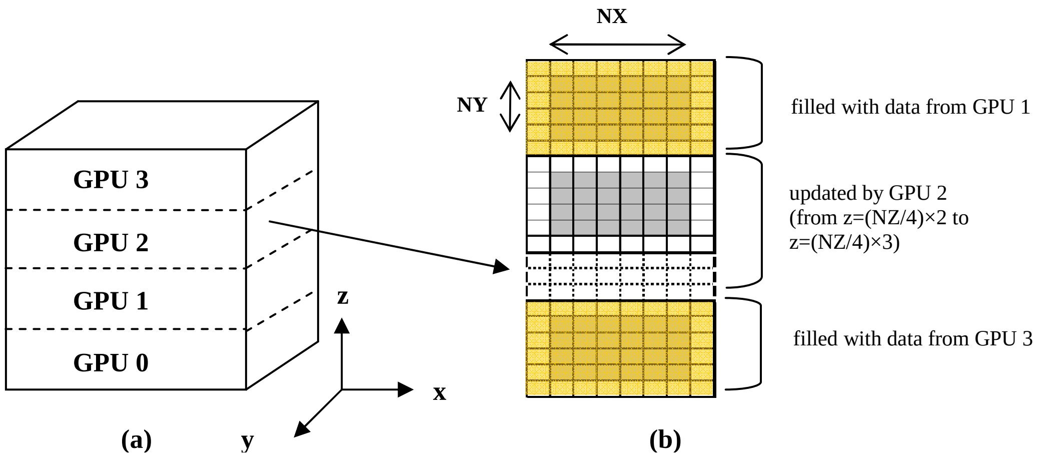

Figure 4. a) Subdomain assignment for multi-GPU solution. b) Representation of the GPU global memory. Each GPU needs ghost cells to represent the top and bottom neighboring cells which are updated by other GPUs (represented here in red).

A. Multi-GPU Implementation

```
//for each time step
for (t=0; t < steps; t++)
{
   //copy velocity ghost cells from host to GPU (top and bottom)
   …
   //call kernel to compute momentum
   momentum <<< grid, block >>>(u, v, w, uold, vold, wold, gpuCount, *device);
   //apply boundary conditions
   momentum_bc <<< grid, block >>>(u, v, w, gpuCount, *device);

   //copy velocity border cells from GPU to host memory(top and bottom)
   …
   //synchronize with other threads before reading updated ghost cells
   pthread_barrier_wait(&barrier);

   //copy velocity ghost cells from host to GPU(top and bottom)
   …

   //call kernel to compute divergence
   divergence <<< grid, block >>>(u, v, w, div, gpuCount, *device);

   //for each Jacobi solver iteration
   for(m = 0; m< njacobi; m++)
   {
     // compute pressure
     pressure <<< grid, block >>>(div, pold, p, gpuCount, *device);
      ptemp = pold; pold=p; p=ptemp;
     pressure_bc <<< gridDims, blockDims >>>(d_p, s_gpuCount, *device);

     //copy pressure border cells from GPU to host memory (top and bottom)
     …
     //synchronize with other threads before reading updated ghost cells
     pthread_barrier_wait(&barrier);

     //copy pressure ghost cells from host to GPU (top and bottom)
     …
   }
   //velocity correction
   correction <<< grid, block >>>(u, v, w, p, gpuCount, *device);
   momentum_bc <<< grid, block>>>(u, v, w, gpuCount, *device);

   //copy velocity border cells from GPU to host memory(top and bottom)
   …
   //synchronize with other threads before reading updated ghost cells
   pthread_barrier_wait(&barrier);

   //rotate matrices
   utemp = uold; uold=u; u=utemp;
   vtemp = vold; vold=v; v=vtemp;
   wtemp = wold; wold=w; w=wtemp;
}
```

Listing 2. Partial host-side code that implements the projection algorithm[30] to solve the Navier-Stokes equations for incompressible fluid flow. The outer loop is used for time stepping while the inner loop is in the iterative solution of the pressure Poisson equation. A CPU thread is created for each available GPU and executes the code above. Synchronization between the CPU threads is done through a Posix *barrier*.

In the multi-GPU implementation, each GPU is responsible for a subdomain of size NX×NY×(NZ/number of GPUs), as shown in Fig. 4a. The whole domain is represented on the host side while the GPUs only store their respective subdomains in global memory, and the ghost cells used to update the cells at the bottom and the top of the subdomain. As shown in Fig. 4b, 2×NX×NY ghost cells need to be filled with data from the GPUs responsible for the top and bottom neighboring subdomains. At the GPU level, the subdomain is mapped to a 2D CUDA grid the same way it was for the single-GPU implementation.

With the domain decomposition shown in Fig. 4a, each GPU needs neighboring data computed by other GPUs which means all GPUs need to synchronize to exchange velocity and pressure fields at each time step. But a GPU cannot directly exchange data with another GPU. Hence, ghost cells at the multi-GPU domain decomposition boundaries needs to be copied back to the host, which adds an extra communication overhead to the overall computation in addition to the CUDA kernel launches at every time step.

As mentioned earlier, multi-GPU parallelism is not currently addressed by CUDA. We assign one CPU thread to each GPU so that each device has its own context on the host. Listing 2 shows the host side code snippet for the multi-GPU implementation of the projection algortihm. Each CPU thread executes the code given in Listing 2. First the GPUs copy the top and bottom cells of their subdomains fom their global memory to a matrix on the host side. After the GPUs are synchronized using a Posix barrier (`pthread_barrier_wait`), the GPUs read from the host-side matrix data that represent their ghost cells and update their global memory. For the velocity field, this process happens twice per time step, once after the solution of the momentum equations and once after the correction step. For the pressure field, data exchange occurs after each Jacobi solver iteration.

V. Kernel Acceleration Using the Shared Memory on the GPU

A. Shared Memory Implementation

Usage of the shared memory (SM) in a kernel is a three-step process. First, the block threads copy the subdomain they are responsible for from the global memory to the shared memory. Then computation is done by the threads, using data from the shared memory. Finally the result of the computation is written back to the global memory before exiting the kernel. This back and forth data transfer between the global memory to the shared memory creates an overhead that is not present in a global memory implementation. Hence, the arithmetic intensity of the kernel should be sufficiently large to compensate for the overhead of data copying in order to benefit from the shared memory implementation. One way to achieve this is to increase the size of the subdomain that is mapped to a thread block. Figure 5 compares two different domain decompositions where each block contains 4×4 threads. In the first one (Fig 5a), the block is directly mapped to a subdomain of 4×4 computational nodes. In order to update those computational nodes, the subdomain and all its surrounding nodes (ghost cells) need to be copied to the shared memory. To update 4×4 cells, 6×6×3 cells actually need to be copied to the shared memory. In which case, less than 15% of the shared memory will be updated by the thread block.

The second approach shown in Fig. 5b allows threads to update multiple cells in a distinct vertical column. In this example each thread works on two cells (one in the red plane and in the orange plane). The threads are now working on 4×4×2 cells and 6×6×4 cells are required in total. The cells to be updated now represent 22% of the data

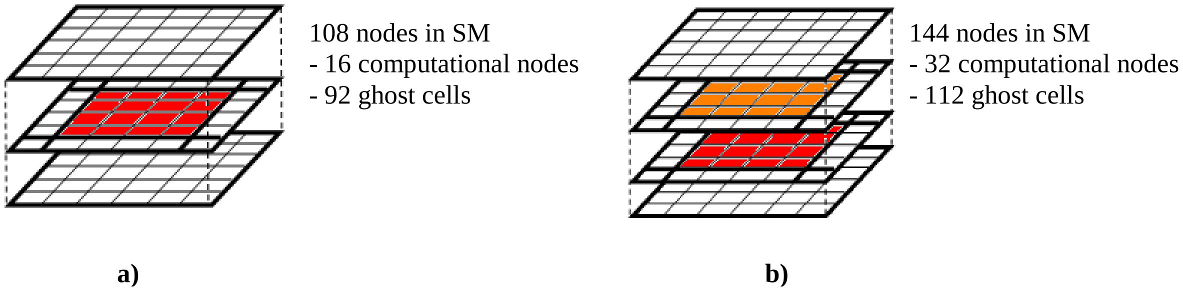

Figure 5. Two different approaches for shared memory usage in a 4×4 block configuration. Colored cells are updated by the threads while the white cells are only used as data source (ghost cells). Each cell center represents a computational node. a) Each thread updates one cell only (red cells). b) Each thread works on 2 cells in the same vertical column. Cells in red are updated during the first iteration and orange ones in the second iteration.

brought to the shared memory. This can be easily implemented by having a *for* loop iterating in the z-direction for each thread. Notice that the number of iterations in the z-direction is known in advance as it is defined by the programmer. The directive `#pragma unroll` can be used to unroll the *for* loop. Then the cost of the *for* loop is not critical while the mapping shown in Fig. 5b reduces the amount of time spent in transferring data from the global to the shared memory.

As the size of the block and the number of cells to update per thread increase, the overhead due to data copying to the shared memory is compensated by the time spent on actual computations. For our current shared memory implementation, each block works on two different levels in the XY-plane. The size of the shared memory being limited to 16 KB, we cannot bring more than four levels (two inner levels and two ghost levels) into the shared memory if the block size is 16×16. Note that if the shared memory gets too large, few threads can be created at the same time as less registers are available[10]. An alternative implementation would be to have fewer threads per block but more levels for each thread to work. Further tests will give us more insight on the optimal configuration.

B. Memory Model Specific Implementation of the Projection Algorithm

The projection algorithm involves distinct steps in a predictor-corrector fashion in the solution of the fluid flow equations. In the current study, each step is implemented as a kernel to be computed on the GPU, as shown in Listings 1 and 2. We have implemented both global and shared memory versions of the kernels needed to implement the projection algorithm on multi-GPU desktop platforms. Figure 6 shows the speedup for the kernels resulting from a shared memory implementation. The speedup is measured relative to a global memory implementation.

Usage of the shared memory in the *momentum* and *pressure* kernels make them perform over 2× faster relative to a kernel implementation that uses only the global memory. These two kernels benefit from the shared memory because the overhead due to data transfer to the shared memory is largely compensated by their high arithmetic intensity. Also in a global memory implementation the numerical discretization scheme leads to non-coalesced memory accesses because the scheme needs data from neighboring cells, which slows down performance. The shared memory implementation avoids this issue by sharing data among threads in the same block. The other kernels (i.e., *correction*, *momentum_bc* and *pressure_bc*) are not presented in Fig. 6, because their arithmetic intensity is either low or non-coalesced memory accesses are not substantial.

Ad-hoc testing of each kernel implementation for computational speedup allows us to suggest a unique implementation of the projection algorithm[30] that exploits the memory architecture of the GPU on desktop platforms. Based on the implementation shown in Listings 1 and 2, we suggest a shared memory implementation for the velocity prediction step and the solution of the pressure Poisson equation, whereas we suggest a global memory implementation for the kernels to compute the divergence field, velocity corrections and impose the boundary conditions.

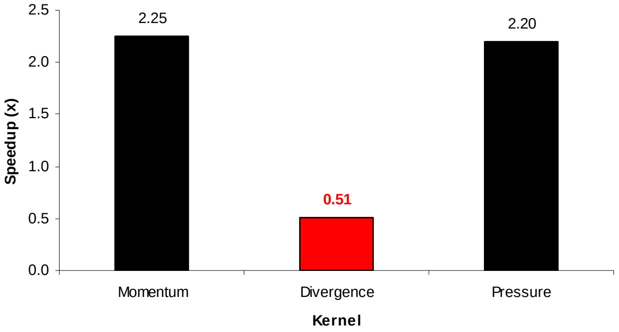

Figure 6. Kernel speedup when using shared memory over global memory (computational domain size is 256×32×256). Tests showed that the momentum and pressure kernels benefit from a shared memory implementation, giving a speedup of more than 2× relative to a kernel implementation that uses only the global memory.

VI. Results and Discussions

A. Validation of the Multi-GPU Implementation of the Navier-Stokes Solver

The lid-driven cavity problem is a well-established benchmark case in the CFD field, and it can be used to validate the implementation of the conservation mass and momentum principles, because it is a closed system. The fluid inside the cavity is driven by the motion of the lid on the top surface. Figure 7a shows the velocity field taken at the middle section in the vertical plane at steady state for Re=1000 based on the lid velocity and cavity height. At this Reynolds number, the flow is laminar and remains two-dimensional. Note that we adopt 3D computations to assess the computational performance of GPUs for large computational problems. Otherwise, the simulation can be

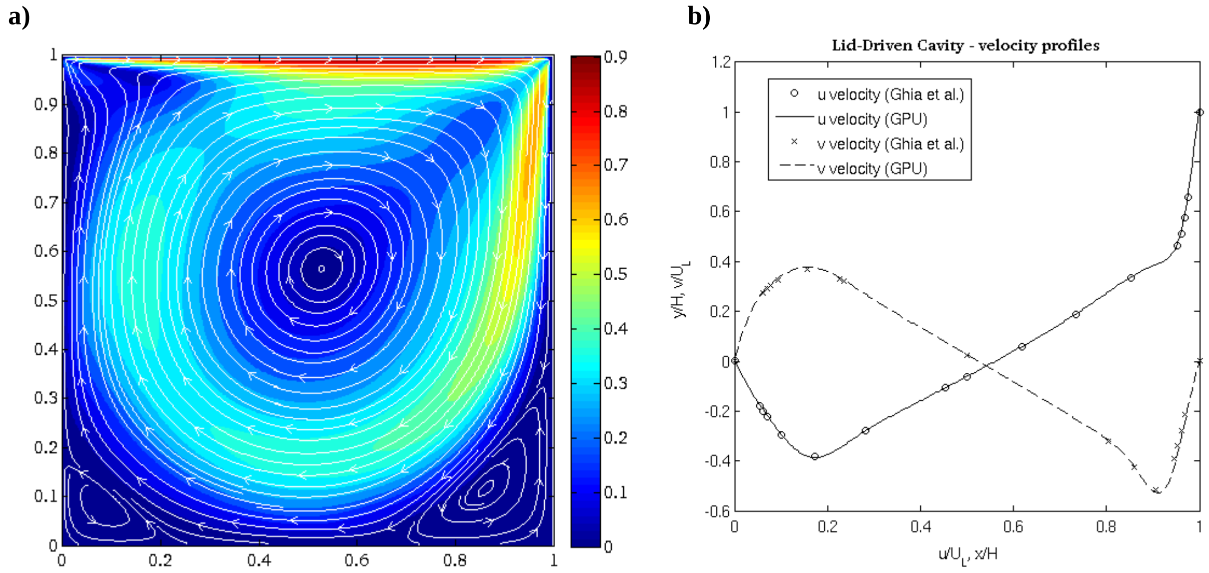

Figure 7. a) Distribution of velocity magnitude and streamlines at steady-state for Re=1000. Low velocity regions are represented in dark blue while high velocity regions are represented in red. b) Comparison of the multi-GPU implementation of our CFD code results with benchmark data given in Ghia et al[28].

performed by 2D computations. Figure7a shows the velocity streamlines at steady-state. The flow structure inside a cavity for various Reynolds numbers is well established. Any mistake in the implementation can be quickly detected by inspecting the streamlines and the distribution of the velocity field. For Re=1000, one should observe a main circulation at the core of the cavity, and smaller recirculation zones at the bottom corners. The size of these corner vortices increases with the Reynolds number. To validate our multi-GPU implementation, results are compared to numerical data from Ghia et al.[28] in Fig. 7b. The present results obtained from the GPU code are in excellent agreement with the results of Ghia et al.[28]

B. Performance Evaluation of the Serial CPU Implementation of the CFD Code

Before proceeding to the GPU speedup assessment, we test the FLOPS performance of our serial CPU implementation of the CFD code against comparable applications. Both the serial CPU version and GPU version of our CFD code adopts the same numerical methods. The NAS Parallel Benchmarks[31] (NPB's) were derived from CFD codes. NPB was designed to compare the performance of parallel computers and it is widely recognized as a standard indicator of computer performance. In Table 1, the LU, MG and SP benchmarks from NPB are compared to our in-house developed serial CFD code programmed in C language. The code was compiled with GNU C Compiler[32] (gcc) using optimization level O3 with CPU architecture specifications (i.e., -march=core2 for Intel Core 2 Duo; –march=opteron for the AMD Opteron). The NPB benchmarks were compiled with the Intel Fortran compiler.

Table 1 shows that the performance of our in-house serial CFD code is comparable to the NPB benchmark codes in terms of Giga Floating Point Operations per Second (GFLOPS). Using only a single core, the performance of our CFD code is approximately 1.6 GFLOPS on the Intel Core 2 Duo 3 GHz, 1.0 GFLOPS on the AMD Opteron 2.4 GHz processors. Interestingly, the GFLOPS performance drops to approximately 0.50 when the computational problem size is substantionally increased. Figure 8 shows more details about the performance of our serial CPU version CFD code with increasing problem size.

Table 1. GFLOPS performance of the serial CPU version of our CFD code and NPB benchmark codes on two different computers (Intel Core 2 Duo (E8400) 3.0 GHz and AMD Opteron (8216) 2.4 Ghz). LU factorizes an equation into lower and upper triangular systems. The iteration loop of MG consists of the multigrid V-cycle operation and the residual calculation. SP is a simulated CFD application. Our CFD code simulates a lid-driven cavity problem.

		GFLOPS		Ratio
Benchmark	Size	*Intel Core 2 Duo 3 GHz*	*AMD Opteron 2.4 GHz*	Intel / AMD
LU.S	12 x 12 x 12	2.52	1.55	1.62
LU.W	33 x 33 x 33	2.54	1.02	2.48
LU.A	64 x 64 x 64	2.13	0.68	3.13
LU.B	102 x 102 x 102	1.20	0.68	1.78
MG.S	32 x 32 x 32	2.35	1.26	1.86
MG.W	128 x 128 x 128	1.64	0.87	1.88
MG.A	256 x 256 x 256	1.67	0.73	2.29
MG.B	256 x 256 x 256	1.78	0.79	2.27
SP.S	12 x 12 x 12	3.00	1.55	1.94
SP.W	36 x 36 x 36	2.36	0.76	3.09
SP.A	64 x 64 x 64	1.46	0.70	2.08
SP.B	102 x 102 x 102	1.38	0.49	2.81
In-house	32x32x32	1.58	1.03	1.54
CFD code	1024x32x1024	1.42	0.54	2.64

On the Intel Core 2 Duo 3 GHz processor, the serial CPU version of our CFD code performs pretty well as the GFLOPS number drops only by 10% when the domain size increases by a factor of 1024 (i.e., domain size increases from 32^3 to 1024^2×32). To put this into context, the SP benchmark performance drops by 54% when the domain size increases by a factor of 614 (i.e., domain size increases from 12^3 to 102^3). Figure 8 shows GFLOPS performance drop on AMD Opteron 2.4 GHz when the domain size gets larger than 128×32×128 (20 MB in memory). This was also observed with the NPB benchmark codes for problem sizes requiring over 20 MB of memory.

The results shown in Fig. 8 and Table 1 indicate that the serial CPU version of our CFD code is fairly optimized, giving performance comparable to NPB benchmark codes. Advance code optimizations techniques may improve the GFLOPS performance of the serial CPU version of our CFD code, but it is not pursued in the present study.

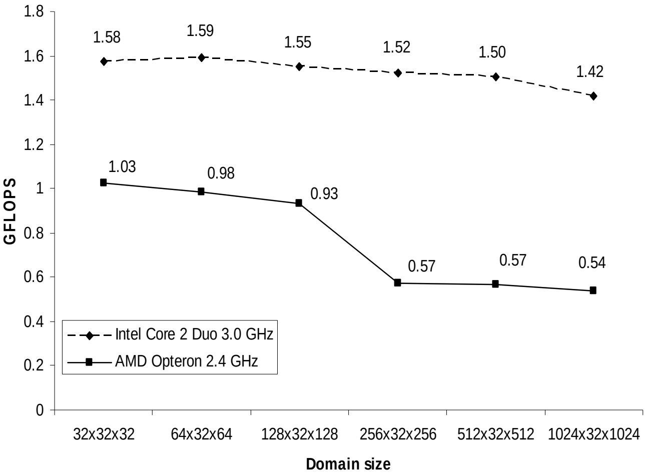

Figure 8. GFLOPS performance of the serial CPU version of our in-house developed CFD code with increasing computational domain size.

C. Performance Results

The following computing hardware was utilized in this study. A dual-CPU/dual-GPU platform was built with an Intel Core 2 Duo 3.GHz (E8400) CPU, 4GB of memory and two Tesla C870 boards. Each Tesla board provides 128 streaming processor cores and 1.5 GB of global device memory. A second platform with 8 AMD Opteron 2.4GHz (8216) dual-core CPUs, and a Tesla S870 server provides 16 CPU cores and four GPUs. Each GPU board used in this study can deliver a theoretical peak performance of 512 GFLOPS[3]. These two high performance computing platforms with different GPU-CPU configurations were used to perform speedup and multi-GPU scaling analysis.

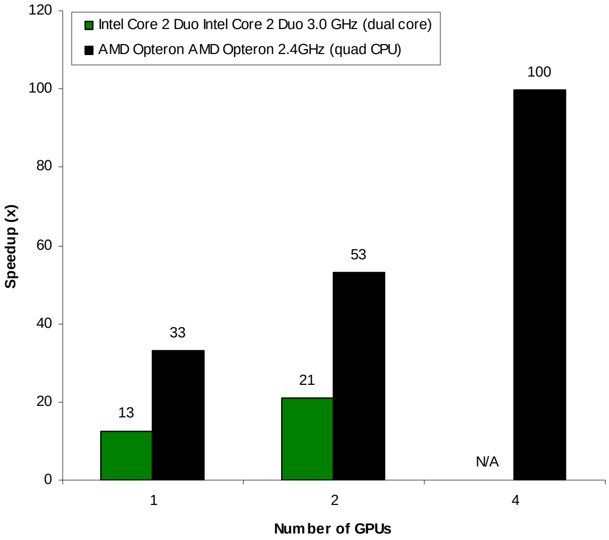

Figure 9. GPU code speedup relative to the serial CPU code for a domain of 1024×32×1024 computational nodes. Quad-GPU results are currently not available for the Intel Core 2 Duo platform, because we do not have the hardware available for the present study.

Using only a single CPU core, the serial CPU version of our CFD code takes 82,930 seconds on the Intel Core 2 Duo 3.0 GHz CPU and 218,580 seconds on AMD Opteron 2.4 GHz CPU to simulate the lid-driven cavity problem with a computational grid of 1024×32×102 for 10,000 time steps. The serial CPU version of the CFD code runs faster on Intel Core 2 Duo CPU than on AMD Opteron CPU because of its larger L2 cache and its better clock frequency. On the other hand the execution time for the GPU code is barely dependent on the CPU clock speed. GPU performance was nearly the same on both the Intel and AMD platforms. As a result GPU performance relative to the CPU performance is better for the AMD Opteron 2.4 GHz platform as shown in Fig. 9. On our Intel Core 2 Duo platform the GPU code performs 13 and 21 times faster than the CPU code with one and two GPUs, respectively. On the AMD Opteron 2.4 GHz platform the GPU code performs 33, 53 and 100 times faster using one, two and four GPUs respectively.

Figure 10 shows computational speedup with respect to different problem sizes. On the AMD Opteron platform (Fig. 10a), depending on the problem size, the quad-GPU performance varies from 10× to 100× relative to the serial CPU version of the CFD code. On the Intel Core 2 Duo platform (Fig. 10b), the dual-GPU performance varies from 5× to 21×. The speedup numbers are impressive for large problem size, because the arithmetic intensity on each GPU increases with problem size, and the time spent on data communication with other GPUs compared to the time spent on computation becomes relatively shorter. For small problems, a multi-GPU computation performs slower than the single-GPU computation. On the AMD Opteron platform, for a problem of size 64×32×64, the dual-GPU solution performs slower than the single-GPU, and the quad-GPU solution performs slower than the dual-GPU (Fig. 10a). As more GPUs are available, the domain treated in the simulation should be larger to have each GPU working on large subdomain, and hide the latency due to GPU data exchange. Based on our current implementation, tests have shown that performance is better when there is a one-to-one matching between the number of GPUs and number of CPUs on desktop platforms. For example, a dual-CPU platform coupled to the quad-GPU S870 server did not show any gain in performance over a dual-CPU dual-GPU Tesla C870 platform. Note that this statement is dependent on our implementation, and performance may be improved by overlapping communication with computation.

Figure 11 shows the multi-GPU performance scaling on the NVIDIA Tesla S870 server. The speedup results shown in Fig. 9a are converted to scaling numbers. By increasing the problem size and adjusting the size of the data to exchange between the GPUs, the performance on the quad-GPU platform is 3× the performance of a single GPU, and the dual-GPU solution performs 1.6× faster than the single GPU. These performance numbers are less than the ideal performance numbers of 4× and 2×, respectively. The bottleneck of the multi-GPU solution is the data exchange between the GPUs, which requires synchronization and data transfer form the different GPUs to the host and vice versa. Pinned memory (or page-locked memory) usage may reduce the time spent in exchanging data between the host and the devices, leading to a better scaling in parallel multi-GPU computations.

VII. Conclusions

We have presented the implementation of Navier-Stokes equations for incompressible fluid flow on desktop platforms with multi-GPUs. NVIDIA's CUDA programming model is used to implement the discretized form of the governing equations. The major steps of the projection algorithm are implemented with separate CUDA kernels, and a unique implementation that exploits the memory hierarchy of the CUDA programming model is suggested.

Kernels for the velocity predictor step and the solution of the pressure Poisson equation were implemented using the shared memory of the device, whereas a global memory implementation was pursued for the kernels that are responsible to calculate the divergence field and velocity corrections and to apply the boundary conditions. This unique combination resulted in factor of two speedup relative to global memory only implementation on the device. To the best of our knowledge, our work is the first implementation of an incompressible flow Navier-Stokes solver on multi-GPU desktop platforms.

Overall, we have accelerated the numerical solution of incompressible fluid flow equations by a factor of 100 using the NIVIDIA S870 Tesla server with quad GPUs. The speedup number is measured relative to the serial CPU version of our CFD code that was executed using a single core of an AMD Opteron 2.4 GHz processor. With respect to a single core of an Intel Core 2 Duo 3.0 GHz processor, we have achieved a speedup of 13 and 21 with single and dual GPUs (NVIDIA Tesla C870), respectively. Same numerical methods were adopted in both the CPU and GPU versions of the CFD code. We have observed that multi-GPU scaling and speedup results improve with increasing computational problem size, suggesting that computationally "big" problems can be tackled with GPU clusters with multi-GPUs in each node. We have also found that in a multi-GPU desktop platform, one CPU core should be dedicated to each active GPU in order to obtain good scaling performance across multi-GPUs.

Our future work will focus on CUDA-specific optimization strategies and adding a complex geometry capability to our multi-GPU parallel CFD code. We also plan to extend our code to address turbulent flow regimes.

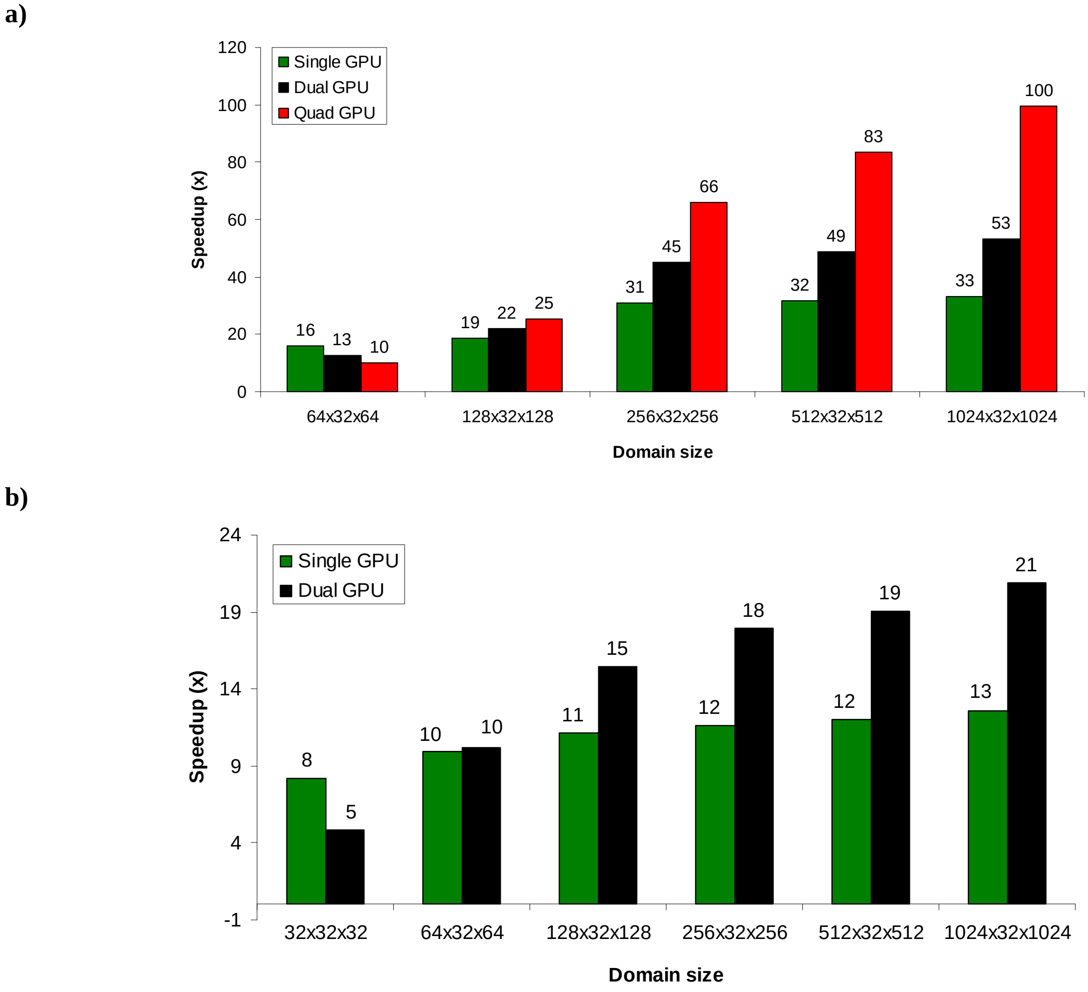

Figure 10. Single and multi-GPU speedup relative to a single CPU core. a) AMD Opteron 2.4GHz with NVIDIA S870 Quad Tesla server b) Intel Core 2 Duo 3.0GHz with dual NVIDIA C870 Tesla boards.

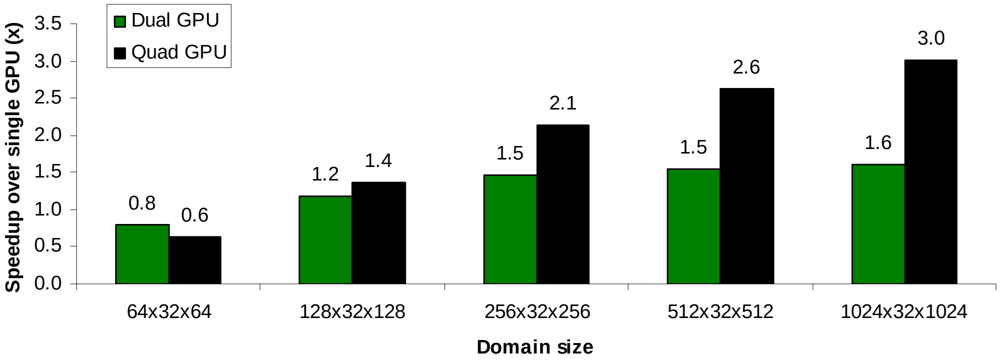

Figure 11. Multi-GPU scaling on the S870 server with quad-CPU platform. As the problem size increases the multi-GPU solutions scale better.

Acknowledgments

The authors thank Drs. Massimiliano Fatica, Patrick Legresley, David Luebke from NVIDIA and Timothy J. Barth from NASA Ames Research Center for helpful discussions on CUDA and GPU computing. Thanks are extended to Marty Lukes and Luke Hindman of Boise State University for their help on building our desktop supercomputer, and to NVIDIA Corporation and Micron Technology, Inc. for hardware donations. This work is partially funded by NASA Idaho EPSCoR Research Initiation grant.

References

[1] Hennessy, J. L., Patterson, D. A., Goldberg, D. and Asanovic, K., *Computer Architecture: A Quantitative Approach*, 4th ed., Morgan Kaufmann, San Francisco, 2006.

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

[3] NVIDIA, “NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0,” 2008.

[4] Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E. and Phillips, J. C., “GPU Computing,” *Proceedings of the IEEE*, Vol.96, IEEE Publishing, 2008, pp. 879-899.

[5] 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, No.3, 2004, pp. 777-786.

[6] Stratton, J. A., Stone, S. S. and Wen-mei, W. H., “MCUDA: An Efficient Implementation of CUDA Kernels on Multi-cores,” IMPACT Technical report, IMPACT-08-01, University of Illinois at Urbana-Champaign, March 12, 2008.

[7] Houston, M., “Stream Computing,” *International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2008 classes*, article 15, ACM, New York, 2008.

[8] The MPI Forum, “The Message Passing Interface (MPI) Standard,” URL: http://www-unix.mcs.anl.gov/mpi/ [cited 29 December 2008]

[9] Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R., *Parallel Programming in OpenMP*, Morgan Kaufmann, San Francisco, 2001.

[10] Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B. and Wen-mei, W. H., “Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA,” *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM, New York, 2008, pp. 73-82.

[11] Harris, M. J., “Real-Time Cloud Simulation and Rendering,” Ph.D Dissertation, University of North Carolina, Chapel Hill, NC, 2003.

[12] Stam, J. “Stable fluids,” *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, 1999, pp. 121-128.

[13] Liu, Y., Liu, X. and Wu, E., “Real-time 3D Fluid Simulation on GPU with Complex Obstacles,” *Proceedings of the Computer Graphics and Applications, 12th Pacific Conference*, IEEE Computer Society, Washington, DC, 2004, pp. 247-256.

[14] Li, W., Fan, Z., Wei, X. and Kaufman, A., “GPU-Based Flow Simulation with Complex Boundaries,” *GPU Gems 2*, Addison-Wesley, Boston, MA, 2005, pp. 747–764.

[15] 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, 2004, p. 47.

[16]Anderson, J., Lorenz, C. and Travesset, A., “General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units,” *Journal of Computational Physics*, Vol. 227, No. 10, 2008, pp. 5342-5359.

[17]Liu, W., Schmidt, B., Voss, G. and Muller-Wittig, W., “Molecular Dynamics Simulations on Commodity GPUs with CUDA,” *Lecture Notes in Computer Science*, *High Performance Computing – HiPC 2007*, Vol. 4873, Springer, New York, 2007, pp.185-196.

[18]Ufimtsev, I. and Martinez, T., “Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-electron Integral Evaluation,” *Journal of Chemical Theory and Computation*, Vol. 4, No. 2, 2008, pp. 222-231.

[19]Schatz, M. C. and Trapnell, C. Delcher, A. L. and Varshney, A., “High-throughput Sequence Alignment using Graphics Processing Units,” *BMC Bioinformatics*, BioMed Central, 2007.

[20]Barrachina, S., Castillo, M., Igual, F. D., Mayo, R. and Quintana-Orti, E. S., “Solving Dense Linear Systems on Graphics Processors,” Technical Report ICC 02-02-2008, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores, February 2008.

[21]Castillo, M., Chan, E., Igual, F. D., Mayo, R., Quintana-Orti, E.S., Quintana-Orti, G., van de Geijn, R. and Van Zee, F.G. “Making Programming Synonymous with Programming for Linear Algebra Libraries”, Technical Report, University of Texas at Austin, Department of Computer Science, Vol. 31, April 17, 2008, pp. 8-20.

[22]Michalakes, J. and Vachharajani, M., “GPU Acceleration of Numerical Weather Prediction,” *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, Washington, DC, 2008.

[23]Bleiweiss, A., “GPU Accelerated Pathfinding,” *Proceedings of the 23rd ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, Eurographics Association, Aire-la-Ville, Switzerland, 2008, pp. 65-74.

[24]Tolke, 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.

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

[26]Molemaker, J., Cohen, J. M., Patel, S. and Noh, J., “Low Viscosity Flow Simulations for Animation,” *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, Eurographics Association, Aire-la-Ville, Switzerland, 2008.

[27]NVIDIA, “CUDA Zone, the resource for CUDA developers,” URL: http://www.nvidia.com/object/cuda_home.html [cited 29 December 2008]

[28]Ghia, U., Ghia, K. N. and Shin, C. T., “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.

[29]Ferziger, J. H. and Peric, M., *Computational Methods for Fluid Dynamics*, Springer, New York, 2002.

[30]Chorin, A. J. “Numerical Solution of the Navier-Stokes equations,” *Mathematics of Computation*, Vol. 22, No. 104, 1968, pp. 745-762.

[31] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D.S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O. Lasinski, T. A., Schreiber, R. S., “The NAS Parallel Benchmarks,” *International Journal of High Performance Computing Applications*, Vol. 5, No. 3, 1991, pp. 63-73.

[32]GCC, GNU Compiler Collection, Ver. 4.1.2, Sept. 2007, URL: http://gcc.gnu.org [cited 29 December 2008]