Proceedings of the ASME 2011 International Mechanical Engineering Congress & Exposition
IMECE2011
November 11-17, 2011, Denver, Colorado, USA

IMECE2011-65260

COMPUTATIONAL FLUID DYNAMICS USING GRAPHICS PROCESSING UNITS: CHALLENGES AND OPPORTUNITIES

S. Pratap Vanka
Dept. of Mechanical Science and Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
spvanka@illinois.edu

Aaron F. Shinn
Dept. of Mechanical Science and Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
afshinn2@illinois.edu

Kirti C. Sahu
Dept. of Chemical Engineering
Indian Institute of
Technology Hyderabad
Andhra Pradesh, India
ksahu@iith.ac.in

ABSTRACT

A new paradigm for computing fluid flows is the use of Graphics Processing Units (GPU), which have recently become very powerful and convenient to use. In the past three years, we have implemented five different fluid flow algorithms on GPUs and have obtained significant speed-ups over a single CPU. Typically, it is possible to achieve a factor of 50-100 over a single CPU. In this review paper, we describe our experiences on the various algorithms developed and the speeds achieved.

INTRODUCTION

This paper describes some of our recent experiences of using Graphics Processing Units as a paradigm for performing large-scale scientific computations. In particular we are interested in computational fluid dynamics (CFD), which is important to a large number of mechanical, aerospace, chemical and biomedical industries. Beginning with essentially no background in using GPUs for CFD, we have, over the years, implemented several methodologies of CFD on a GPU, and studied a number of flow problems. The objective of this paper is to provide our assessment of learning, implementing, and applying the codes to problems of our interest. The currently observed performances are sufficiently impressive and attractive to pursue this new paradigm as a tool for CFD. Further code optimizations and tuning of the data structures may permit further speed-ups. It is also necessary to mention here that the technology is continually improving, and new hardware platforms as well as software are being developed. Hence many of the experiences reported here are being quickly superseded with new products and compilers being released by GPU vendors such as NVIDIA. Our current experiences relate to NVIDIA GPUs (specifically the Tesla C1060 and C2070) and programming them using CUDA (Compute Unified Device Architecture).

DESCRIPTION OF GPU ARCHITECTURE

The GPU can be thought essentially as a massively parallel computer, capable of simultaneously executing instructions on a large number of arithmetic units. However, because of the special architecture of the GPU, it is necessary to devise the numerical algorithm as well as the program structure such that the communication and computation as well as data access are executed optimally. The architecture of a GPU is quite different than that of a CPU. A GPU is designed with more transistors dedicated to computation and less resource dedicated to data caching and flow control compared with a CPU, resulting in significant computational speed-up [1]. The GPU is designed to be a parallel processor by using massive multithreading, where a single thread can be thought of as the smallest unit of execution that executes instructions in a program. Instructions for the GPU are written in a “kernel" which is similar to a function in the C programming language.

When a kernel is executed on a GPU, each thread executes the statements in that kernel, where each thread maps to a different element of data. Thus, the GPU architecture can be classified as SIMD (single-instruction, multiple-data) or SIMT (single-instruction, multiple-thread [1]). The number of threads needed for a particular kernel depends on the data size to be processed, since the threads map to the data element indices. Threads are organized into “blocks," and all blocks belong to a “grid" as shown in Fig. 1. Before a kernel is executed on a GPU, the dimensions of the blocks and grid must be set explicitly by the programmer, as these are not automatically set by the GPU. It should be noted that the GPU is used as a co-processor in conjunction with the CPU.

Typically, the “main" program executes on a CPU, and the GPU is utilized by launching kernels from the main program. Thus, usage of the CPU is not eliminated but rather is minimized. A GPU contains multiprocessors, where each contains streaming processors or “cores." For example, the Tesla C1060 has 30 multiprocessors, each with 8 streaming processors, and thus has a total of 240 streaming processors. The streaming processors are responsible for processing the thread blocks. When a block is processed, the threads in the block are divided into groups of threads (called warps), and the streaming processor launches the threads in a warp in parallel [1]. The blocks are independent of each other and there is no synchronization among blocks, so the only way to ensure all blocks have executed is to wait until the kernel has finished and control has been returned to the main program.

In addition to the architectural differences between CPUs and GPUs, the memory bandwidth is another important difference. Modern GPUs have memory bandwidths an order of magnitude greater than CPUs; this is due to CPUs having to satisfy constraints of legacy applications and operating systems, which makes increasing memory bandwidth difficult, whereas GPUs have less legacy constraints resulting in more memory bandwidth [2]. This increase in memory bandwidth is another factor contributing to the favorable performance of GPUs.

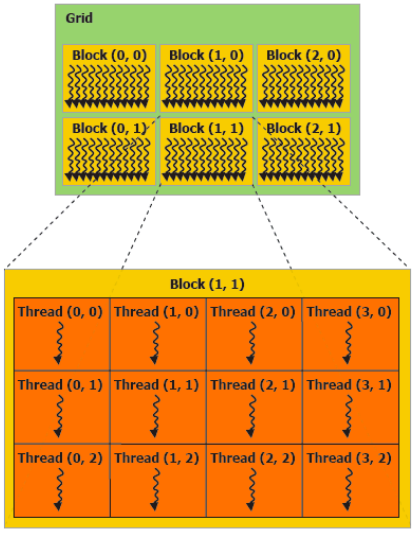

FIGURE 1. THREAD HIERARCHY OF THE GPU [1].

The memory spaces (RAM) of the CPU and GPU are separate, and explicit copy operations must be performed to move data to/from the GPU. Global memory is the largest memory space on the GPU (4 GB for Tesla C1060, 6 GB for Tesla C2070) but it is not cached, and thus has long access times. Each thread has its own registers and local memory, which are used for storing local variables declared within the kernel. All threads on a given block have access to the block's shared memory. However, a thread on a given block cannot access another block's shared memory.

Shared memory is cached, and can be accessed much faster than global memory, but it is limited in size. In order to use shared memory, data must first be transferred from global memory to shared memory; computations are then performed using shared memory and the results are written back to global memory. This introduces extra computational complexity in the algorithm, but this can be offset by the potential gains of using a cached memory space. This benefit is realized if the data loaded into shared memory are reused many times during kernel execution. Texture memory is a read-only cached memory space that can be accessed by all threads, which offers avenues for performance optimization. For example, data structures in global memory can be read through texture memory via texture fetching, which decreases the access time. The Tesla C2070 (Fermi architecture) also has L1 and L2 caches. While limited in size, these caches provide another avenue for rapid memory access.

BRIEF OVERVIEW OF SELECTED PREVIOUS WORKS

The use of GPUs for CFD applications is rapidly getting popular, and a number of researchers have found this paradigm to be beneficial. With the availability of small clusters of GPUs, performances of several tens of teraflops are possible on low footprint and low energy consuming “supercomputers”. A variety of scientific applications have been programmed on GPUs by a number of researchers, and their references can be found on several websites, especially at the NVIDIA website. Here we present some earlier works, and some recent works concerning CFD applications on GPUs. This list is by no means meant to be complete. In addition, note that comparisons to single-core CPU simulations can exaggerate the GPU speedup. Comparisons to multi-core CPU implementations would be better (although the present authors admit that we did not adhere to this practice since a multi-core CPU version of our codes was not developed).

Before the advent of CUDA, programmers had to cast their applications in terms of graphics processing operations. Early work of this type was done by Scheidegger et al. [3], where they presented a GPU implementation of the SMAC method (Simplified Marker And Cell) to solve the 2D incompressible Navier-Stokes equations on structured grids. Central differences and a hybrid donor cell scheme were used in their approach. Texture memory was used to store the data structures; for example, floating-point textures called pixel buffers (or “pbuffers") were used to store the velocity fields. The Jacobi iteration scheme was used as a fragment program for the solution of the pressure-Poisson equation. In their study, two GPUs were tested: a GeForce FX 5900 (NV35) and a GeForce 6800 Ultra (NV40). The CPU used was a 2 GHz Pentium IV. The CPU performed better than the GPU only when a very small mesh was used, and this occurred when using the NV35. They explain that convergence is rapid in this case and that the pbuffer switches “probably overshadowed” the GPU parallelism. Their approach was, on average,

approximately 16 times faster than the CPU version. They studied a variety of flows, such as a lid-driven cavity, rising smoke at high Reynolds number, and flow past the outline of a car at low Reynolds number.

Elsen et al. [4] used a GPU to simulate the inviscid flow in simple and complex geometries by numerically solving the compressible Euler equations. Compared to the CPU, they achieved GPU speed-ups of over 40 times for simple geometries and 20 times for complex geometries. The complex geometries consisted of a NACA 0012 airfoil and a hypersonic vehicle at Mach 5. Their comparisons were based on a 2.4 GHz Intel Core 2 Duo (single core used) and NVIDIA 8800GTX. They used the Navier-Stokes Stanford University Solver (NSSUS), which is capable of solving the 3D Unsteady Reynolds Averaged Navier-Stokes (URANS) equations. This code uses the finite-difference method with a vertex-centered solution on multi-block meshes. Temporal evolution of the solution toward a steady-state was accomplished using an explicit five-stage Runge-Kutta scheme. The code also incorporated a geometric multigrid scheme to accelerate convergence. For their study, only the steady solution of the compressible Euler equations was sought. They used the BrookGPU language to implement NSSUS (which was originally in Fortran) on a GPU.

Brandvik and Pullan [5] presented results for 2D and 3D Euler solvers implemented on the GPU. Their original implementation on the CPU of the Euler solvers was written in Fortran. They used the Euler solvers to simulate turbine flows: the 2D solver was used to simulate the flow through a transonic turbine cascade and the 3D code was used to simulate secondary flow development in a low speed linear turbine cascade. The 2D solver was programmed for the GPU using the BrookGPU language, and performed 29 times faster than the CPU version. They also used BrookGPU for the 3D solver, which performed only 3 times faster than the CPU version. A CUDA implementation of the 3D solver yielded better performance with a speed-up of 16 over the CPU. A 2.33 GHz Intel Core 2 Duo processor was used for the CPU solvers, where only a single core was utilized. The 3D CUDA solver used an NVIDIA 8800 GTX graphics card and the 2D and 3D BrookGPU solvers used an ATI 1950XT graphics card. Their 2D and 3D codes solved the compressible Euler equations using the finite volume method with structured grids, where the variables were stored at the cell vertices. The spatial derivatives were discretized via second-order central differences and the temporal derivatives were discretized to first-order accuracy. There was no multigrid method employed in their approach.

In another work by Brandvik and Pullan [6], they present a three-dimensional Navier-Stokes solver implemented on multiple GPUs using MPI. Instead of implementing in a particular language targeted at a particular hardware, they instead generalized their solver by expressing the subroutines in the Python scripting language. They developed a source-to-source compiler used to convert these subroutines into source code to be compiled for a given target architecture (multi-core CPUs, NVIDIA GPUs, etc.). This novel approach clearly has the advantage of making the code more flexible and it provides longevity to the code since it will be easier to adapt to future architectures.

Cohen and Molemaker [7] present a GPU implementation using CUDA for solving the incompressible Navier-Stokes equations with the Boussinesq approximation. They present results for the simulation of the Rayleigh-Benard convection problem and compare their GPU implementation to a multithreaded Fortran solver running on an eight-core CPU. Using double precision, the GPU-based solver was approximately eight times faster. Shinn and Vanka [8] were the first to implement the SIMPLE algorithm on a GPU. Using CUDA, they wrote a 2D solver with multigrid Full Approximation Scheme (FAS) used to accelerate convergence of all flow variables (u, v, and p). The code was tested for the benchmark 2D driven cavity problem and compared to a CPU version of the code written in Fortran. It was found that the speedup scales with the problem size. For a problem size of 512 x 512 grid cells, the GPU was an order-of-magnitude faster than the CPU for a range of Reynolds numbers. Steady-state calculations of driven cavity flow with 4096 x 4096 could be performed in a minute of GPU time.

Shinn et al. [9] performed one of the first Direct Numerical Simulations using GPU hardware. The fractional-step method with finite volume spatial discretization was used to solve the incompressible Navier-Stokes equations, and was implemented on a GPU using CUDA. A geometric multigrid method was used to accelerate the pressure-Poisson solution. They simulated turbulent flow in a square duct at a bulk Reynolds number of 5480 using a mesh resolution of 26.2 million cells. This problem was selected not only to validate the GPU-based solver but also to test the capability of the GPU, as this was the largest problem that could fit on a single Tesla C1060. The salient features of this canonical flow were captured and compared well with previous data. The GPU-based solver was over an order of-magnitude faster compared with the CPU-based version. Chaudhary et al. [10] extended this solver to include magneto-hydrodynamics and used it to study the magnetic field effects on turbulent flow in a square duct. Direct Numerical Simulations were performed at a bulk Reynolds number of 5500 at different Hartmann numbers to vary the magnetic field.

Thibault and Senocak [11] presented the first implementation of a 3D incompressible Navier-Stokes solver on multiple GPUs. Using CUDA, a fractional-step procedure was used to solve the equations and the pressure-Poisson equation was solved using Jacobi iteration with no multigrid scheme. The spatial terms were discretized with second-order accurate central differences and an explicit, first-order accurate Euler scheme was used for temporal advancement. They validated their GPU implementation and assessed speedup via the problem of laminar flow in a lid-driven cavity.

Griebel and Zaspel [12] were the first to implement a two-phase Navier-Stokes solver on a GPU, where they used a level set technique for the two-phases and a fractional step method to solve the Navier-Stokes equations. They implemented the solver on multiple GPUs and communicated GPU data between CPUs using Message Passing Interface (MPI). They ported a solver for the pressure-Poisson equation (a Jacobi-preconditioned conjugate gradient solver) and the level set reinitialization to the GPU using CUDA. In order to minimize the overhead from data communication, they exploited the asynchronous communication feature of CUDA, where data can be copied while computations are being performed. This can effectively hide the communication time.

This was done using "streams" where one stream managed communication while the other managed computation.

Other than finite-difference and finite-volume methods, there has been progress in implementing the Lattice-Boltzmann Method (LBM) for simulating fluid flows on GPUs. Early work by Li et al. [13] provided an implementation of the LBM on a GPU. Their implementation was capable of dealing with complex boundaries (both moving and deformable), which were managed using a voxelization algorithm. All calculations were performed on a GPU in real time and their simulations were second-order accurate in time and space. Their LBM code was programmed in Cg and OpenGL and used an NVIDIA GeForce FX 5900 Ultra GPU. The CPU used was a 2.53 GHz Pentium IV. They simulated a number of complex geometries on the GPU, such as a vase, a sphere, and a swimming jellyfish. It was found that their GPU implementation was 8 to 15 times faster than the CPU counterpart.

Tolke [14] used the 2D Lattice Boltzmann Method on a GPU by programming in CUDA. The implementation was tested by simulating the fluid flow through a porous medium, which consisted of a grid of 324 circular cylinders, equally spaced in the horizontal and vertical directions. The GPU implementation was over 10 times faster relative to the CPU. Peng et al. [15] developed a 3D Lattice Boltzmann Method algorithm for a GPU using CUDA. They compared an NVIDIA GPU (GeForce 8800 GTS) with an AMD CPU (Sempron 3500+) and found that the GPU performed 8.76 times faster than the CPU. As an example of a complex geometry, they used their LBM implementation for the simulation of fluid flow through fractured glass.

Recently, Marsh [16] used CUDA to implement a hybrid molecular dynamics/Lattice Boltzmann Method on GPUs. Flow through a nano-scale straight channel and a nanoscale bellow channel were investigated. In the hybrid method, a molecular dynamics solver was used in the near-wall region and a Lattice Boltzmann solver was used away from the wall. The GPU provided a speed-up factor of 5-10 for the molecular dynamics solver and 50-75 for the Lattice Boltzmann solver compared to a CPU. The large speed-up for the Lattice Boltzmann method is indicative of the fact that this method is easier to parallelize (or, strictly speaking, multithread) compared to molecular dynamics. As an extension of this work, Sahu and Vanka [17] implemented a two-phase LBM on a GPU and observed a speed-up factor of 25 over a CPU.

IMPLICATIONS OF GPU FOR CFD

As mentioned above, several researchers have ported/developed numerical algorithms on GPUs. In order to take full advantage of the speeds offered by GPUs, a number of modifications have to be made to any existing CFD legacy code. In many situations, the algorithm/code may have to be rewritten specifically suited to GPUs, otherwise the maximum possible speed is not achieved. Here we describe some of our experiences. First, explicit time-marching algorithms are the most convenient ones to be ported on to the GPU. This is because there is no iteration, and the new value of a variable depends only on the old time values. Hence, the update of a given variable can be done independent of variables being updated on other threads. There is no recursive relation between the variables on the threads, since they are all known at the old time step. However, even for explicit algorithms, a few changes may be needed for efficiently implementing on the GPU. These relate to the use of shared memory and the layout of data structures. Memory coalescing and block size influence the speed achieved. Memory coalescing is guaranteed if the data is accessed such that sequential threads access sequential nodal data. With the Fermi architecture, the requirements to achieve coalescing are more relaxed [18]. In addition, data should be, where possible, copied to shared memory and re-used as much as possible. Threads belonging to the same block can make use of the shared memory for that block which can sometimes be used to enhance the algorithm efficiency by reducing the number of global memory accesses.

Even explicit algorithm based CFD codes need to be reorganized to take advantage of the GPU architecture. When an implicit algorithm is used, the efficiency as well as the convergence is impacted. Implicit algorithms directly ported to a GPU will not work because of the mixed implicit and explicit updates. It is necessary to remove any recursive updates, so the algorithm can be run on parallel threads. As an example, consider the Gauss-Seidel algorithm. Because of the recursive relation, it is necessary to "color" the nodes such that values of one color are not related within themselves. For a five point stencil of a 2D Poisson equation, two colors will generate sets in which each variable is not connected to its own members. Each color is then processed sequentially. However, one should not use the modulo operator to skip nodes of a different color. That would waste the threads, and also the data are not consecutively placed. Instead, one must reorganize the data to obtain the best and most use of the threads and memory. For higher-order stencils, more colors will be needed, and that may complicate the code structure. For line inversions, which are also recursive, it is necessary to have a second dimension along which the lines can be organized in colors. Two colors for a second-order stencil can be generated in which lines of cells are not connected with each other. Thus, each line, though recursive within itself, can be solved on one thread.

Lattice Boltzmann algorithms are the easiest ones to develop on the GPU because of their inherent data parallel nature. A Lattice Boltzmann algorithm consists of three steps: collision, streaming, and calculation of flow variables. The collision step and calculation of flow variables are very much local operations. They can be performed independently on all threads. However, it is necessary to select the number of lattice points, and the layout such that the block size is optimal, and also the threads access adjacent data. This can be done by "un-rolling" the density function vector, and writing one array for each of the components. This will increase the program length, but can bring efficiency. The streaming step, where the density function is advected to the neighbor lattice points, is the "tricky" part. Here, there is no computation, and the step requires pure data replacement. Use of shared memory is advantageous here, in which chunks of data are simultaneously copied from and to the global memory. Lattice Boltzmann algorithms have also been extended to two-phase flows, but require calculation of derivatives of some functions. These derivative calculations require values at neighbor locations, and can degrade performance. Our recent observation has been that single phase algorithms with combined collision and equilibrium calculations can achieve a speed-up of 50-75 over a CPU, whereas two-phase algorithms run slower (only about 25 times faster than a CPU). These speed differences are however

a function of other variables such as CPU compiler, hardware, and also multiple or single core CPU.

OUR RECENT RESEARCH

Recently, we have implemented five different algorithms on the GPU, and compared their performance with those of a CPU. In most cases, the speed-ups have been very attractive. Large-scale CFD calculations with up to 25 million nodes could be performed on a single Tesla GPU and in real compute times that are fairly competitive with a supercomputer facility. In this section, we describe these efforts and present both the computational performance and the computed flow problems.

Fractional Step Method for DNS/LES of Turbulent Flows

Our first code developed on a GPU was for conducting direct and large eddy simulations of turbulent flows. Such DNS/LES are very computationally intensive, requiring massive amounts of storage and CPU time. We have considered only incompressible flows, for which a fractional step algorithm has been used. In this algorithm, the momentum and energy equations are solved by an explicit algorithm with second-order temporal and spatial accuracy. The finite volume method with a staggered grid is used. As the momentum equations are updated explicitly, no iterations are required, and there are no recursive steps. However, if an implicit formulation is used either for all convective and diffusive fluxes, or just for diffusive fluxes, a special algorithm is needed. The most time consuming step is the pressure-Poisson equation, which is fully implicit. The pressure-Poisson equation requires convergence to a high degree (mass error), and consumes nearly 80% of the total time. In our serial method, we have used Successive Over-Relaxation (SOR) which has a better convergence rate than a pure explicit Jacobi scheme. However, since the grids used are very fine, we have accelerated this using geometric multigrid on a structured grid. Several levels of finite volume grids nested within a fine grid are used. The traditional SOR is not parallelizable; hence we have used a red-black coloring scheme [19] to separate the unknowns in two independent subsets. The mesh is "colored" like a checkerboard and the red cells are updated, then the black cells (or vice-versa). The multigrid is implemented with a V-cycle, and consists of restriction, relaxation and prolongation.

The solution of the pressure-Poisson equation is done to a high accuracy, typically three or four orders of magnitude reduction in error at every time step. Our current implementation, uses a modulo operator, where the threads that are red are skipped when the black colored cells are solved and vice-versa. In an effort to decrease memory access times in the SOR implementation, textures were used to fetch the pressure data from global memory, which decreased overall code execution time by approximately 10 percent. Also, we explored using shared memory in the SOR algorithm but did not see much benefit, either due to low data reuse or a sub-optimal implementation. Global memory was used for all other array accesses in the other kernels.

To understand how GPU threads map to computational cells, consider Fig. 2, which shows a mesh of the internal cells with dimensions of nx[level] x ny[level] x nz[level]. The arrays nx[level], ny[level], and nz[level] contain the number of mesh cells in each direction for the given mesh level in the multigrid V-cycle. The indices of the internal cells range from (i, j, k) = (2, 2, 2) to (i, j, k) = (nx[level]+1, ny[level]+1, nz[level]+1). The boundary cells (which are not shown in Fig. 2) lie along the planes i = 1, j = 1, k = 1 and planes i = nx[level] + 2, j = ny[level] + 2, k = nz[level] + 2. The GPU grid dimensions are (gx, gy, gz) and each block has dimensions (bx, by, bz). The GPU grid dimensions gx and gy were calculated by dividing the dimensions of the computational mesh on the current mesh level by the block size. Thus, while performing multigrid, the GPU grid dimensions are changed to accommodate the size of the current computational mesh level. This idea is shown in the example code of Fig. 3, where the execution configuration in the main program (on the CPU) is changed as a function of the mesh level when calling a kernel for the GPU. This example is for the "down-leg" of a V-cycle, where the grid levels start at the finest level (n=1) and descend to the coarsest level (n=ngrid).

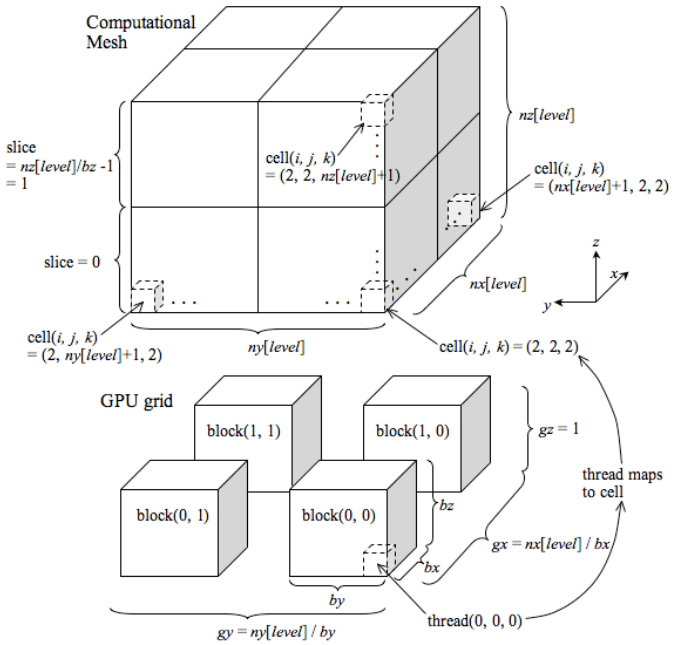

FIGURE 2. CORRESPONDANCE BETWEEN GPU GRID AND COMPUTATIONAL MESH.

The GPU grid and computational mesh have the same dimensions in the x- and y-directions, so that the threads map one-to-one with the cells. However, due to the fact that the GPU grid can only have a z-dimension equal to one (gz=1) requires the threads to be reused for other cells in that direction. This is done by operating on slices of the computational mesh, where a thread for an (i, j) location updates one cell in each slice. Thus a single thread operates on multiple cells, moving in the k direction in a column for fixed (i, j). No threads are assigned to the boundary cells, since no updating is performed there.

The mapping concept shown in Fig. 2 is implemented in the kernel code shown in Fig. 3. The thread indices (tx, ty, tz) are computed from the built-in GPU variables threadIdx, blockIdx, blockDim, which are the thread index in a given block, block index of a given block, and block dimension of a given block, respectively. The thread indices always start at zero, so they are incremented by two in order to map them to

the computational mesh indices. In order to update all the cells in the mesh, a loop that goes over all slices was used inside the kernel to allow a thread for an (i, j) location to update one cell in each slice. As the slice index varies in the loop, so does the k index for the cells that the thread operates on. Also, note that the (i,j,k) indices are mapped to a single cell number "m". This is used to access the data stored in 1D arrays. The array "begin" contains the beginning cell number for each grid level for multigrid, and is used as an offset in the mapping to access a given grid level.

```
 int main(void)
{
  ...
  for(n = 1; n<=ngrid; n++)
 {
  dim3 block(bx,by,bz);
  dim3 grid(nx[n]/bx,ny[n]/by);
  kernel<<<grid, block>>>(...);
 }
 ...
} // end main

 __global__ void kernel(...)
{
 // global thread indices
 tx = threadIdx.x + blockIdx.x *  blockDim.x;
 ty = threadIdx.y + blockIdx.y *  blockDim.y;
 tz = threadIdx.z;
 // convert thread indices to mesh indices
 i = tx + 2;
 j = ty + 2;
 for(slice=0; slice<=nz[n]/blockDim.z-1;
     slice++)
 {
  k = tz + slice * blockDim.z + 2;
  m = i + (j-1)*(nx[n]+2) +
    (k-1)*(nx[n]+2)*(ny[n]+2) + begin[n] - 1;
  ...
  computations
  ...
 } // end slice
} // end kernel
```

FIGURE 3. EXAMPLE OF KERNEL CODE USED IN SOLVER.

Performance is very sensitive to block size, so this is another area of code optimization. Block sizes must evenly divide into the mesh dimensions for each mesh level for multigrid. A block size that can accommodate the coarsest level could be selected, which would accommodate all finer mesh levels. However, this may not yield optimal GPU performance since the block size is small (smaller than the warp size). A compromise between accommodating each mesh level and performance was found by using two block sizes: a block size for the finer meshes and a block size for the coarser meshes. Most of the computation occurs on the finer meshes (first one or two mesh levels in the V-cycle), and thus the block sizes for these levels were tuned for optimal performance. It was found that for most problems a good block size is (bx, by, bz) = (32, 1, 8). This can change from problem to problem, so it is best to experiment to determine the optimal sizes. For the coarser meshes, a smaller block size was used so that the mesh resolution would be evenly divisible by the block size. The smaller block size delivers poor performance, but this only occurs on the coarse levels, which do not have appreciable computing times, so the effect is small.

The performance of the solver on a CPU (written in Fortran) versus on a GPU (written in CUDA) is compared for two different problems in Tables 1 and 2. The CPU was a 2.6 GHz AMD Phenom quad-core processor (single core used) and the GPU was a Tesla C2070 (Fermi architecture). The CUDA 3.2 compiler was used for the GPU executables and the gfortran compiler with the -02 optimization was used for the CPU executables. Table 1 shows the simulation performance of laminar flow in a lid-driven cavity at a Reynolds number of 1000 based on the lid speed and cavity edge length. Table 2 shows the simulation performance of DNS of turbulent flow in a square duct at a Reynolds number of 360 based on the friction velocity and hydraulic diameter.

TABLE 1. PERFORMANCE FOR SIMULATION OF LAMINAR FLOW IN LID-DRIVEN CAVITY. TIMINGS TAKEN FOR FIRST 100 TIME-STEPS OF SIMULATION.

mesh	CPU time (seconds)	GPU time (seconds)	speedup (CPU/GPU)
16x16x16	0.46	0.34	1.35
32x32x32	4.49	0.82	5.48
64x64x64	46.15	2.84	16.25
128x128x128	420.20	17.38	24.18

TABLE 2. PERFORMANCE FOR DNS OF TURBULENT FLOW IN A SQUARE DUCT. TIMINGS TAKEN FOR FIRST 100 TIME-STEPS OF SIMULATION.

mesh	CPU time (seconds)	GPU time (seconds)	speedup (CPU/GPU)
128x32x32	27.63	2.03	13.61
256x64x64	275.96	12.76	21.63
512x64x64	569.04	24.53	23.20
512x128x128	1997.05	97.26	20.53

Lattice Boltzmann Method for Two-Phase Flows

A second example we show here is the implementation of a two-fluid LBM. In a single phase LBM, only one set of equations for the density function is collided and streamed, then the flow variables are evaluated from this density function and its moments. In the two-fluid LBM, two density functions are collided and streamed, then are used to compute the flow variables. One of the density functions is used to compute an interface variable, which gives the fluid density. The second function gives the velocity and pressure field. This method is based on the method of He et al. [20]. We had applied this

method earlier, implemented on a CPU, to study buoyancy-driven flow in a tilted channel. It is now implemented on a GPU, with a speed-up factor of 25 over a 3.2 GHz single core CPU. A further increase in speed is possible by unrolling the density function array in nine (or 27) individual arrays, as shown recently by Kuznik et al. [21].

Multigrid Acceleration of the SIMPLE Algorithm

The SIMPLE algorithm solves the two- (and three-) dimensional fluid flow equations using an implicit relaxation procedure. In contrast with a time-marching procedure such as the fractional step method, the SIMPLE algorithm solves directly for the steady state (or quasi-steady) flow fields by iteratively updating the velocities and pressure fields. The coupled equations are solved sequentially by updating the velocities using the momentum equations and the pressure field using a pressure correction equation derived from the continuity and truncated momentum equations. The SIMPLE algorithm solves the discrete equations over the complete flow domain in a decoupled manner using single (or multigrid) iterative procedures. Both staggered and collocated arrangements of the flow variables have been used, the latter arrangement utilizing a momentum-interpolation procedure to avoid checkerboard pressure splitting. The computational steps in the SIMPLE algorithm are as follows:

Begin iterations

a) Solve x-momentum equation over entire flow domain
b) Solve y-momentum equation over entire flow domain
c) Compute mass residuals in the momentum velocities
d) Solve pressure-correction equation to annihilate the mass residuals
e) Update the velocities and pressures based on pressure corrections computed in step (d)
f) Solve other scalar transport equations, if any

Repeat steps (a) to (f) until convergence of all equations is acheived. Steps (a), (b), (d) and (f) involve the solution of a set of linear equations with coefficients linking a local value with its neighbors. The set of linear equations can be solved iteratively with single or multigrid versions of standard iterative solvers such as the Gauss-Seidel or Thomas algorithm. The system of equations is usually diagonally dominant and positive definite if appropriate discretizations are employed.

The multigrid method has been included inside SIMPLE at two levels in the algorithm. First, the linear solver used to solve the set of discrete linear equations with a given set of coefficients can be accelerated by using the multigrid method. This resolves the low frequencies in the linear equations, providing good feedback between the momentum and continuity equations. However, it does not resolve efficiently the low frequency errors in the coupling between the momentum and continuity equations. A second stage of multigrid acceleration that provides the most benefit and resolves the low frequency errors in the inter-equation coupling is over the entire iterative sequence. Here we have incorporated the MG method for the entire sequence of equations using the Full Approximation Scheme (FAS), which is suited to nonlinear equations.

Table 3 presents timings for the multigrid procedure implemented on the CPU. We present results for different Reynolds numbers and different mesh sizes. Here we have used the traditional Gauss-Seidel iterative scheme with a fixed number of sweeps and fixed number of V-cycles. The computer times are presented together with the convergence levels achieved.

Before timing tests are performed on the GPU, an optimization study was conducted to determine the best block size to use when calling a GPU kernel. GPU performance is sensitive to the block size, and thus is important to tune for maximum performance. To achieve a good convergence rate, the grid must be sufficiently coarsened in the V-cycle, and here we take the coarsest grid to be 4x4. To accommodate the 4x4 mesh we take a block size of 4x4 and a GPU grid size of 1x1 (the grid has only one block). However, this block size is by no means optimal, as will be shown. Thus for finer meshes we use a better block size to achieve better performance, and on coarser meshes we default to using the 4x4 block size. Here we take "fine mesh" to mean a mesh that is evenly divisible by the better block size, and any mesh that is not is considered a "coarse mesh" and uses the 4x4 block size. A performance study was conducted using a variety of block sizes for the "fine mesh"; the "coarse mesh" block size was held fixed at 4x4. The result of this study is presented in Table 4, where a 1024x1024 mesh with 9 grid levels at Re=100 was used as the test case. It was found that the 32x1 block size performs best, which is not too surprising since there are 32 threads in a "warp" which is the maximum number of threads that can be launched in parallel in a given block at one time.

Finally, in Table 5, we present the performance of the multigrid procedure on the GPU. The optimal block size of 32x1 was used. Since the convergence rate can be different on the GPU and on the CPU, we have performed a fixed number of V-cycles on both CPU and the GPU. The levels of convergence achieved are given for different Reynolds numbers and mesh sizes. In addition, we have performed calculations with the biggest grids possible on the GPU. Our present code requires 24 arrays on the GPU, and for a 4 GB Tesla C1060 processor we have been able to perform calculations on a 4096 x 4096 grid (with 11 levels). The computer times required for such a large problem are indeed small and very attractive even for more complex practical flows.

CONCLUSIONS

Graphics Processing Units have recently evolved as a new paradigm for scientific computations. They are essentially multi-core machines with a large number of compute units sharing a common memory. They can be viewed as single instruction multiple data computers. Their cost/performance ratio, and low power consumption makes them attractive for high-resolution fluid flow computations. However, in order to exploit the inherent architecture of the device, the numerical algorithm, as well as data structures must be carefully tailored to minimize the memory access and any recursive relations in the algorithm. In the past three years, we have developed five different CFD algorithms, and have found speed-ups over a CPU of factors between 10-25, with a possibility of another factor of four gain through optimization. This makes GPUs very attractive for computing industrial fluid flows. However, porting legacy codes automatically is not easy.

TABLE 3. MULTIGRID CPU TIMES FOR ERROR < 10^{-3}.

		Re=100			Re=400			Re=1000			Re=2000		
mesh	levels	time (s)	ncyc	error	time (s)	ncyc	error	time (s)	ncyc	error	time (s)	ncyc	error
64x64	5	2.60E-2	6	3.31E-4	5.50E-2	12	6.59E-4	9.30E-2	20	7.84E-4	0.12	25	7.43E-4
128x128	6	0.14	7	4.66E-4	0.22	11	7.01E-4	0.49	25	7.88E-4	0.71	36	8.19E-4
256x256	7	0.71	8	9.93E-4	0.97	11	5.85E-4	2.03	23	7.92E-4	3.53	40	8.65E-4
512x512	8	5.19	11	5.31E-4	5.66	12	5.46E-4	9.90	21	7.30E-4	17.46	37	7.51E-4

TABLE 4. EFFECT OF BLOCK SIZE ON GPU PERFORMANCE FOR 1024x1024 MESH WITH 9 GRID LEVELS AT Re=100.

	fine mesh		coarse mesh	
GPU time (s)	*bx*	*by*	*bx*	*by*
22.93	1	4	4	4
9.13	4	1	4	4
5.26	4	4	4	4
22.66	1	8	4	4
5.27	8	1	4	4
4.13	8	8	4	4
3.27	16	1	4	4
3.40	16	16	4	4
3.19	32	1	4	4
3.29	64	1	4	4

TABLE 5. MULTIGRID GPU TIMES FOR ERROR < 10^{-3}.

		Re=100			Re=400			Re=1000			Re=2000		
mesh	levels	time (s)	ncyc	error	time (s)	ncyc	error	time (s)	ncyc	error	time (s)	ncyc	error
64x64	5	2.23E-2	6	5.11E-4	4.66E-02	12	7.92E-4	7.65E-2	20	9.47E-4	9.63E-2	25	9.65E-4
128x128	6	4.11E-2	7	5.73E-4	6.44E-02	11	8.90E-4	0.15	25	9.06E-4	0.21	36	9.60E-4
256x256	7	0.11	8	8.05E-4	0.15	11	7.69E-4	0.32	23	9.18E-4	0.55	40	9.94E-4
512x512	8	0.48	11	3.69E-4	0.52	12	8.06E-4	0.91	21	9.48E-4	1.61	37	9.79E-4
1024x1024	9	1.91	12	9.13E-4	2.07	13	7.54E-4	3.67	23	9.42E-4	5.57	20	9.40E-4
4096x4096	11	55.75	18	9.45E-4	52.66	17	7.69E-4	99.15	32	9.88E-4	139.42	45	9.98E-4

A significant rewrite of the algorithm and the code may be necessary. While this may be a hurdle to cross, the time investment may be worthwhile because multi-core architectures of one form or the other are going to be the necessary trend for high resolution / high performance computing.

REFERENCES

[1] NVIDIA, 2010, CUDA C programming guide, version 3.2.

[2] Kirk, D. B. and Hwu, W. W., 2010, Programming massively parallel processors: A hands-on approach, Morgan Kaufmann Publishers, Burlington, MA.

[3] Scheidegger, C. E., Comba, J. L. D., and da Cunha, R. D., 2005, "Practical CFD simulations on programmable graphics hardware using SMAC," Computer Graphics Forum, 24(4), pp. 715-728.

[4] Elsen, E., LeGresley, P., and Darve, E., 2008, "Large calculation of the flow over a hypersonic vehicle using a GPU," J. Comput. Physics, 227(24), pp. 10148-10161.

[5] Brandvik, T. and Pullan, G., 2008, "Acceleration of a 3D Euler solver using commodity graphics hardware," 46th AIAA Aerospace Sciences Meeting.

[6] Brandvik, T. and Pullan, G., 2009, "An accelerated 3D Navier-Stokes solver for flows in turbomachines," ASME Turbo Expo 2009.

[7] Cohen, J. M., and Molemaker, M. J., 2009, "A fast double precision CFD code using CUDA," 21st International Conference on Parallel Computational Fluid Dynamics.

[8] Shinn, A. F., and Vanka, S. P., 2009, "Implementation of a semi-implicit pressure-based multigrid fluid flow algorithm on a graphics processing unit," Proceedings of the ASME 2009 IMECE.

[9] Shinn, A. F., Vanka, S. P., and Hwu, W. W., 2010, "Direct numerical simulation of turbulent flow in a square duct using a graphics processing unit (GPU)," 40th AIAA Fluid Dynamics Conference.

[10] Chaudhary, R., Vanka, S. P., and Thomas, B. G., 2010, "Direct numerical simulations of magnetic field effects on turbulent flow in a square duct," Phys. Fluids, 22(7), pp. 1-15.

[11] Thibault, J. and Senocak, I., 2009, "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," 47th AIAA Aerospace Sciences Meeting.

[12] Griebel, M. and Zaspel, P., 2010, "A multi-GPU accelerated solver for the three-dimensional two-phase incompressible Navier-Stokes equations," Comput. Sci. Res. Dev., 25(1-2), pp. 65-73.

[13] Li, W., Fan, Z., Wei, X., and Kaufman, A., 2005, "GPU-based flow simulation with complex boundaries," GPU Gems II, Chapter 47, Addison Wesley.

[14] Tolke, J., 2010, "Implementation of a Lattice Boltzmann kernel using the compute unified device architecture developed by NVIDIA," Computing and Visualization in Science, 13(1), pp. 29-39.

[15] Peng, L., Nomura, K., Oyakawa, T., Kalia, R., Nakano, A., and Vashishta, P., 2008, "Parallel Lattice Boltzmann flow simulation on emerging multi-core platforms," Euro-Par2008 - Parallel Processing, Lecture Notes in Computer Science, vol. 5168, pp. 763-777, Springer Berlin / Heidelberg.

[16] Marsh, D., 2010, "Molecular dynamics-Lattice Boltzmann hybrid method on graphics processors," Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL.

[17] Sahu, K., and Vanka, S. P., 2011, "A multiphase Lattice Boltzmann study of buoyancy-induced mixing in an inclined channel," submitted to Computers and Fluids.

[18] NVIDIA, 2011, CUDA C programming guide, version 4.0.

[19] Yavneh, I., 1994, "On red black SOR smoothing in multigrid," SIAM J. Sci. Comput, 17(1), pp. 180-192.

[20] He, X., Chen, S., and Zhang, R., 1999, "A Lattice Boltzmann scheme for incompressible multiphase flow and its application in simulation of Rayleigh-Taylor instability," J. Comput. Physics, 152(2), pp. 642-663.

[21] Kuznik, F., Obrecht, C., Rusaouen, G., Roux, J.J., 2010, "LBM based flow simulation using GPU computing processor," Computers and Mathematics with Applications, 59(7), pp. 2380-2392.