GPU Computing for Atmospheric Modeling

Experience with a Small Kernel and Implications for a Full Model

Modern graphics processing units offer enticing speedup potential for atmospheric modeling. However, accelerating such computationally expensive models using GPUs isn't straightforward. Porting a large atmospheric model's most expensive routine to an Nvidia 9800 GX2 let one researcher explore GPU use and glimpse the possibilities for accelerating an entire model.

Today, graphics processing units (GPUs) possess many times the peak compute power of CPUs and are quickly becoming accepted as viable alternatives in domains with compute-intensive workloads. Given their high peak performance and inexpensive cost compared to traditional HPC systems, GPUs are an attractive platform for atmospheric science applications. Researchers have made considerable advances in coding key atmospheric modeling routines—such as Navier-Stokes solvers[1,2] and the Advection Diffusion Equation[3]—for GPUs, often achieving large order-of-magnitude speedups in their applications. Other successful efforts have used GPUs to achieve smaller, but meaningful, speedups of atmospheric models by accelerating a portion to the full code.[4] However, it's more daunting to port an entire application, where the work is spread across a large amount of code, and to achieve significant speedups for the complete model. Here, I discuss some of the barriers to doing this, first through the experience of porting an expensive routine from a large atmospheric model to a GPU, and then by exploring the feasibility of accelerating the entire model in the same manner.

1521-9615/10/$26.00 © 2010 IEEE
Copublished by the IEEE CS and the AIP

Rory Kelly
National Center for Atmospheric Research

The Community Atmosphere Model

The Community Atmosphere Model[5] is one of the most heavily used models at the National Center for Atmospheric Research (NCAR). CAM is a global atmosphere model used for climate and weather research. It's also the primary atmospheric component of the Community Climate System Model (CCSM),[6] which simulates the planet's atmosphere and entire climate system, including land surface, ocean, and sea ice. At NCAR, more compute cycles are spent running CCSM than any other single model, with CAM being directly responsible for many of those cycles. Given the CAM code's importance, my team at NCAR decided to investigate what it would take to accelerate it using GPUs.

CAM consists of roughly 139,000 lines of code, the vast majority of which is Fortran 90. CAM simulates numerous physical processes; among the most computationally expensive of these are radiative processes involving the transmission, emission, reflection, and absorption of various light wavelengths in different atmospheric layers. The most expensive routine in CAM is called RADCSWMX, which approximates the effects of shortwave radiation on the Earth's atmosphere. The RADCSWMX routine divides the solar spectrum into 19 intervals from 0.2 microseconds to 5.0 μm. Within each interval, it calculates how light is scattered and absorbed by molecules in the atmosphere, clouds, aerosols, and the surface. It also computes how light is transmitted and reflected within the atmosphere; this is the most expensive portion of RADCSWMX and is handled by the RADDEDMX subroutine. It is this subroutine that we chose to implement on the GPU.

The RADDEDMX Subroutine

The CAM model subdivides the atmosphere into a collection of 3D cells using a latitude-longitude grid on the surface and extending columns from the grid lines up through the atmosphere. The model further divides the columns vertically into multiple layers between the surface and the top of the atmosphere. The RADDEDMX routine computes reflectivities and transmissivities within each column layer, from the top layer to the surface, using the Delta-Eddington[7] approximation at each layer. The routine requires a relatively large number of input and output fields: for each level in each column, RADDEDMX takes 19 input values and produces 10 output values. A single branch performs additional calculations for column layers containing clouds. For the data set used here, an average of 419 floating-point operations occur in each call, most of which are arithmetic operations (including divisions) along with a few exponentials.

The columns are logically organized into groups (*chunks*), and each chunk's number of columns is a tunable parameter. For each chunk that has columns on the globe's sunlit half, the RADCSWMX routine calls RADDEDMX 19 times (one call for each of the 19 spectral frequency bands). Each time the routine is called, it computes solutions for each layer of each column in the chunk. The solution within each column is independent of those in the other columns, so the problem is embarrassingly parallel over columns. A typical CAM run might have anywhere from thousands to hundreds of thousands of columns, and each column generally contains 18 to 60 vertical layers, with 26 layers being the most common configuration. RADDEDMX uses one layer more than the CAM's configuration for calculations at the top of the atmosphere. For our test case, we chose a standard mid-resolution configuration with 32,000 columns, each with 26 vertical levels. Because the RADDEDMX calculation is the same for each spectral window, we considered only one.

GPU Implementation

The GPU we used for our experiment was Nvidia GeForce 9800 GX2. The 9800 GX2 board actually has two GPUs, each with 16 multithreaded streaming multiprocessors (SMs) and 512 Mbytes of global DDR memory. Each SM contains eight scalar processor (SP) cores, two special function units, and 16 Kbytes of local memory. The card communicates with the host system over a PCIe x16 link. Our host system supported only PCIe 1.0, which can provide 4.0 Gbytes per second of bandwidth between the GPU and the AMD Opteron host system. Figure 1 shows a block diagram of a GPU on the GeForce 9800 GX2.

The CAM model subdivides the atmosphere into a collection of 3D cells using a latitude-longitude grid on the surface and extending columns from the grid lines up through the atmosphere.

We implemented the RADDEDMX routine for the GPU using Nvidia's C language extensions for its Compute Unified Device Architecture (www.nvidia.com/object/cuda_home.html). CUDA programs are written to use massive numbers of threads that execute in parallel. CUDA threads are organized into blocks, which are organized into a grid. Every thread has a unique index in a block, and every block has a unique index in the grid. By combining thread and block indices, threads can be given a unique global index in the CUDA application. Data parallelism is exploited by mapping thread indices to data elements and having multiple threads carry out the same computation on different data in parallel.

CUDA code that's executed on the GPU is written in a special *kernel* function. At runtime, the programmer sets the number of threads in a block and the number of blocks in the grid; the

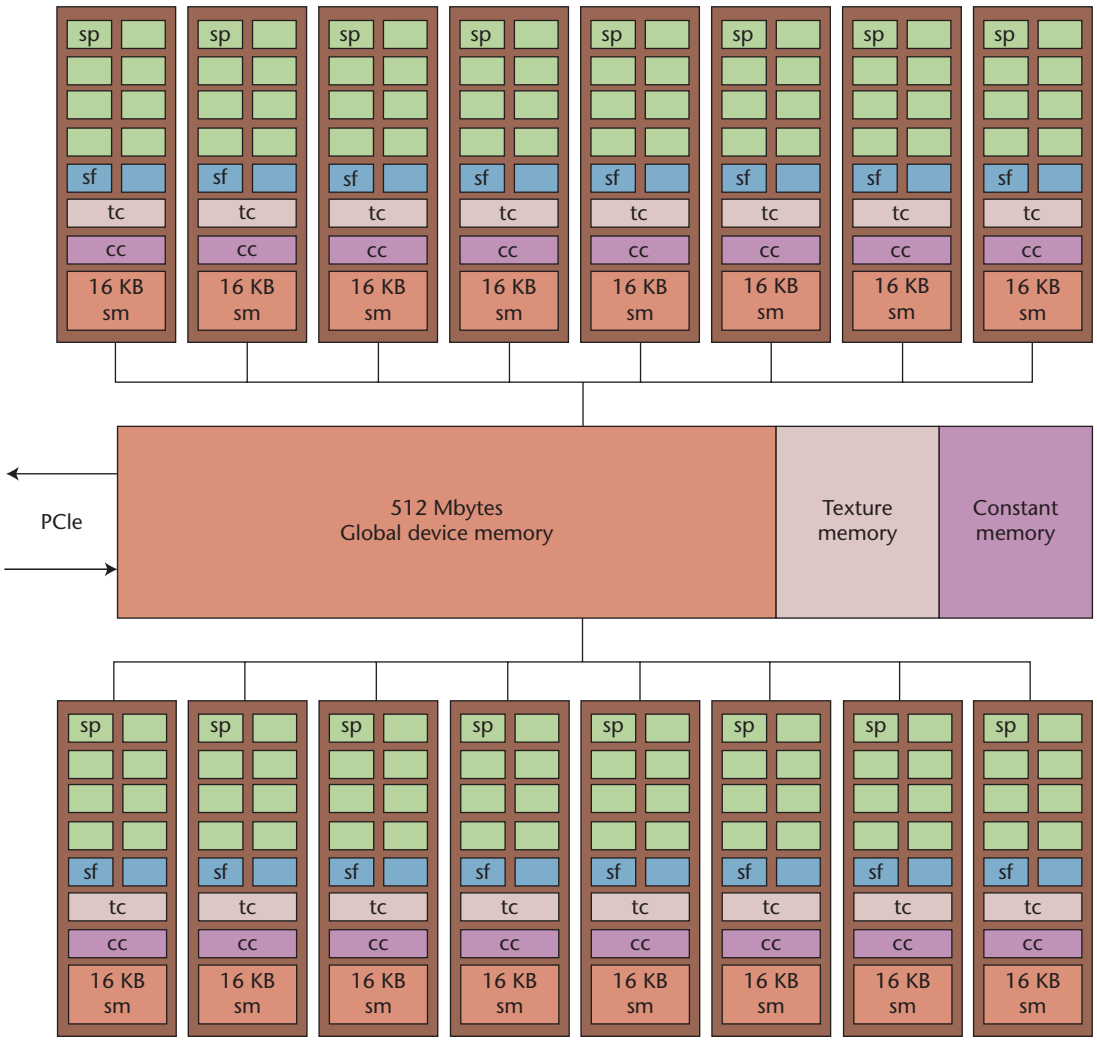

Figure 1. Major components of a G92 core on the GeForce 9800 GX2. The central portion shows global, texture, and constant memory. Each of the 16 boxes above and below corresponds to a streaming multiprocessor (SM). Within each SM, the eight green boxes are scalar processors, and the two blue boxes are special function units. The pink box is 16 Kbytes of local memory. The beige and purple boxes are caches for texture and constant memory, respectively.

kernel is then launched on this grid. Each thread block is scheduled to execute independently on the SMs in the GPU hardware. All threads have access to global device memory, while threads in the same block have access to a faster shared local memory. Optimizing memory access is critical to CUDA performance. Care must be taken to minimize global memory transactions and, when possible, ensure that they're coalesced (combined with other threads' transactions).

Within the CAM model, the `RADDEDMX` computation is performed using double-precision floating-point numbers. The GeForce 9800 GX2 doesn't support double-precision arithmetic, so we implemented the CUDA version of the routine in single precision. For comparison we also used a single-precision version of the routine on the CPU, which allows better use of the available SIMD hardware. More recent Nvidia hardware versions, such as the Tesla line, support double precision, although at a performance penalty. Future products, such as the Nvidia Fermi, promise to further close the gap between single- and double-precision performance.

Because CAM is a Fortran 90 code, and because Nvidia doesn't currently support a Fortran implementation of CUDA, we ported the original `RADDEDMX` routine from Fortran to C. We also created a standalone driver program to call the `RADDEDMX` routine's C version and validate the results using column data taken from a full run of the CAM application. The code's standalone

Column 0 level 0	Column 1 level 0	...	Column $N-1$ level 0	Column 0 level 1	Column 1 level 1	...	Column $N-1$ level 1	...	Column $N-1$ level 27

Figure 2. Layout of variables in the CUDA version's GPU memory. All data arrays are 1D and contain data for adjacent columns in contiguous memory locations for each vertical level. This arrangement guarantees memory coalescing on the GPU.

version let us create a working C version of the routine, which we could then transition to CUDA on the GPU. It also eased the tasks of debugging and checking the code's numerical accuracy. Once the C version of `RADDEDMX` was working and producing the same answers as the Fortran version, we used it to create a CUDA implementation.

The CPU code stores data in 2D arrays by vertical level and column index for the test case's 27 levels and 32,000 columns. The CUDA version uses 1D arrays with data from adjacent columns contiguous in memory on each vertical level. The full arrays are direct-memory accessed (DMAed) to the GPU, where they're stored in a global device memory, the CUDA kernel is called once, and the results are DMAed back to the CPU. The arrays are divided between CUDA threads, with one thread assigned to each column level (that is, $27 \times 32{,}000$ threads). The threads are dispatched in 1D blocks; the number of threads per block is a tunable parameter, and the total number of blocks is automatically adjusted to fit the problem size. Figure 2 shows this data arrangement, which causes neighboring threads to access contiguous memory locations, allowing coalesced memory access and maximizing effective memory bandwidth. Threads can access local memory much faster than global memory, so at the beginning of the kernel call, each thread copies the data it needs into local memory, performs its local calculation, and writes the result back to the correct location in global memory—an important optimization, and a common design pattern in CUDA. After all threads have written their results to the arrays in the global device memory, they're DMAed back to the CPU and the kernel execution is complete. Aside from the extra copies to efficiently manage memory, the kernel's code is very similar to the original C, with just a few function calls traded for faster GPU intrinsics.

Performance

We compared performance for the `RADDEDMX` algorithm's CUDA version against a few common microprocessors. Figure 3 shows the results. Because the `RADDEDMX` algorithm is embarrassingly parallel across columns, scaling among multiple

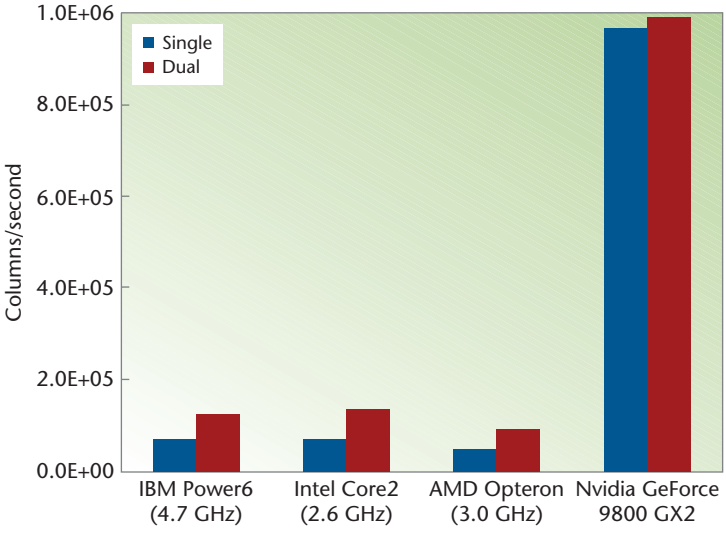

Figure 3. Comparing performance on one and two cores for traditional microprocessors and the Nvidia 9800 GX2 using one and both GPUs on the card. The GPU achieves a speedup of 14 to 20 times in the single-core case and a 7.5 to 11 times speedup in the dual-core case.

CPUs should be nearly linear. The figure's performance results are for one and two respective units on each architecture—that is, one and two multicore CPU cores versus one and two multi-GPU-card GPUs. The fastest CPU result is for the Intel Core2 Duo, which can execute the `RADDEDMX` routine at more than 132,000 columns per second (columns/s), or about 1.5 Gflops/s using both cores. Using only one of the two GPU cores on the 9800 GX2 card, the routine's CUDA version can execute at just over 967,000 columns/s or about 10.9 Gflops/s, including time required to DMA data back and forth between CPU and GPU. Performance using both GPUs isn't much better at 989,000 columns/s (11.2 Gflops/s).

The small performance improvement when using both GPUs is because of the shared PCIe link, which is already saturated in the one-GPU case. Still, the GPU achieves a speedup of 14 to 20 times in the single-core case and a 7.5 to 11 times speedup in the dual-core case. When comparing performance of the CPU and GPU code versions, we must account for the amount of time it takes to transfer data before and after the kernel call. Table 1 shows timings for a single GPU, with and

Table 1. Comparing single core results for the Intel Core2 and the NVIDIA GPU, with and without direct-memory access (DMA) transfer time.

	Intel Core2 2.6 GHz	Nvidia GPU (compute + DMA)	Nvidia GPU (compute only)
Time (ms)	462.64	33.08	2.02
Rate (Gflops/s)	0.782	10.93	179.09
% of peak	7.52	1.42	46.64

without the DMA time included. Peak performance is 10.4 Gflops/s for the Intel Core2 and 768 Gflops/s for the GPU, based on single precision.

Most of the GPU's total execution time is actually spent transferring data over the PCI bus, rather than in the computation itself. For the CUDA's GPU implementation, the total time required to transfer input data, compute results, and transfer output back to CPU memory is 33 milliseconds. Of that 33 ms, 31 ms is spent in transfers, and only 2 ms in computation. If we ignore the cost of data transfer and look at compute rates alone, the GPU can execute the routine with a throughput rate of 15.8 million columns/s—more than 100 times faster than the fastest CPU result. While this reassures us that the kernel is executing efficiently on the GPU, it also highlights the impact severity of data transfer on GPU performance. When writing code to execute on a GPU, it's critical to account for the data transfer penalty and to try and structure the computation to avoid frequent transfers.

The host system we used supported only PCIe 1.0 speeds, so our data transfer was limited to 4.0 Gbytes/s. When we accounted for PCIe protocol overhead and memory system effects, 3.0 to 3.5 Gbytes/s is a more typical DMA transfer rate over the PCIe bus, depending on the transfer size. Our test problem involves a relatively small transfer of roughly 103 Mbytes (69 Mbytes of input and 34 Mbytes of output), which takes around 31 ms total. This gives an effective bandwidth of roughly 3.3 Gbytes/s, which indicates that we're making good use of the limited bandwidth available. To show how constrained the overall problem is by the PCIe bus, we can calculate the maximum compute rate possible with this bandwidth. In the limit of free computing, the data transfer alone takes 31.1 ms, for an execution rate of 1,029,000 columns/s. Our implementation achieved 989,000 columns/s, or roughly 96 percent of the maximum possible performance given the bandwidth constraint. Replacing the host system with one that supports the PCIe 2.0 protocol should nearly double the bandwidth. Provided that the memory subsystem was fast enough to saturate a PCIe 2.0 bus, such a change would considerably impact overall performance.

The Bigger Picture

Overall, our `RADDEDMX` GPU version was successful and achieved meaningful performance increases compared to the original CPU implementation. However, it's important to look at the bigger picture and what this experience implies for using GPUs to speed up the entire CAM model by porting some or all of it to a GPU. Along these lines, three key questions warrant consideration.

What is the Impact of Porting this Subroutine on the Full Model?

For a program to be a good candidate for GPU implementation, it should be dominated by a few intense kernels, with much of the runtime concentrated in a small amount of code. The achievable speedup for a given application when ported to a GPU depends on the percentage of the code that can be sped up on a GPU.

Say, for example, you hope to achieve a 10-times speedup for an application in which 95 percent of the runtime comes from 5 percent of the source code (for CAM, that's still about 7,000 lines). If you port that 5 percent to a GPU and speed it up by 20 times, the complete application will be sped up by 10 times. If you have an application in which work is spread more evenly throughout the code, you'll have to port many more lines of code to the GPU to speed up the full application by the same amount.

This is a problem for many large scientific models, and CAM is a good example because it suffers from a flat performance profile. `RADCSWMX` is the single most expensive routine in CAM, yet it accounts for only 10.9 percent of the total runtime on one processor. `RADDEDMX`, which we implemented on the GPU, consumes only 3.6 percent of the runtime, but is the third most expensive routine in the entire model. As such, even though we managed a respectable speedup of around 14 times for `RADDEDMX`, the amount it would speed up the full model is almost negligible—just over 3 percent. In fact, the 20 most expensive routines combined account for only 48 percent of the total execution time, and every routine beyond that accounts for less than 1 percent of the remainder. So, even if we could port all of the top 20 routines and speed them up by a factor of 1,000, it still wouldn't be sufficient to double the full model's performance. To find 95 percent of the execution

time in CAM, we'd have to include almost all of the source code. Thus, any significant speedups to the full model can come only from successfully porting many tens of thousands of source lines to the GPU and getting them to run extremely well.

Is the RADDEDMX Routine Representative of the Larger Code?

When considering the bigger issue of getting the entire CAM model to run on a GPU, it's important to understand the code's different types of algorithms and consider how well they map to a GPU architecture. RADDEDMX runs efficiently when we rewrite it as a CUDA kernel, but what does that indicate for the rest of the model? One of the RADDEDMX kernel's critical performance factors is how well suited the routine is to running on a GPU. The fact that it's embarrassingly parallel with respect to the columns makes it the ideal case for a GPU. But how much of the rest of the model is this true for? Although some code portions have patches of independent calculations—particularly in the physics—much of the model doesn't fit this pattern. Many parts of the code contain loop dependencies, stencil calculations, shared data, complex branches, and so on. GPUs aren't incapable of handling these situations, but we must pay special attention to memory accesses, synchronization, and efficient data layout.

This problem isn't specific to CAM; it's a trait that many large scientific models share. Codes with a large amount of trivial data parallelism, such as that in RADDEDMX, seem to be the exception, at least in the climate and weather communities. Although this problem isn't insurmountable, efficient implementation of much of the existing scientific code would require more care than we needed for RADDEDMX.

What Aspects Would Be Unmanageable for a Complete Model?

Many aspects of the work we describe here are fine for a small-scale experiment but don't scale well for porting a large model to a GPU system. One such issue is the language barrier. The CAM model is written almost entirely in Fortran, and (at least for the moment), CUDA supports only a C syntax. We thus had to rewrite the RADDEDMX routine into C, and then convert the C into CUDA for the GPU. This is fine for routines that are several hundred lines of code; having to completely rewrite an application of hundreds of thousands of lines, however, isn't a viable option.

This problem would be greatly eased by having either robust translation tools to convert Fortran source to CUDA or by having a CUDA compiler that supported Fortran syntax. Fortunately, several efforts are underway to ease this problem, including

- the US National Oceanic and Atmospheric Administration's free Fortran-to-CUDA accelerator compiler (F2C-ACC; see www-ad.fsl.noaa.gov/ac/F2C-ACC.html);
- Caps Enterprise's commercial Heterogeneous Multicore Parallel Programming compiler (HMPP; see www.caps-entreprise.com/hmpp.html); and
- Portland Group's commercial PGI accelerator compilers (see www.pgroup.com/resources/accel.htm).

Even Nvidia says it will eventually support Fortran for CUDA. All of these tools are either relatively new or haven't seen full release yet. We expect that over the next few years they'll grow rapidly in maturity and capability, and should greatly ease the path to GPU acceleration for a large amount of scientific code.

Many aspects of the work we describe here are fine for a small-scale experiment but don't scale well for porting a large model to a GPU system.

A related issue is that large applications need to maintain a single-source version of the model. For a large code with a small kernel that dominates performance, it might be easy to maintain several versions. In a model like CAM, however, where most of the model must run on GPUs to be effective, it might require too much software engineering time to either maintain multiple-source versions or co-mingle source versions with pragmas and preprocessor directives. Even the Portland Group compiler, which supports a directive-based syntax similar to OpenMP, isn't similar enough that it can share identical source code. We need a better way to generate code for multiple diverse targets from a single shared source. Efforts along these lines exist, including languages such as OpenCL (www.khronos.org/opencl), but it's too early to judge how successful or widely adopted these efforts will be.

Finally, when trying to port the entire CAM model, the code would have to run across many

GPUs in multiple nodes. Clusters aren't going away, they're just becoming more complicated. Many systems coming online in the next few years will resemble traditional clusters, with some or all of the nodes containing GPUs. A test case like ours would typically be run across tens to hundreds of CPUs. If, for example, our case ran on eight nodes of a hypothetical machine with 16 CPU cores and four GPUs in a node, then the 32,000 columns would be shared such that each CPU core received 250 columns. Such a small number of columns wouldn't be enough to fully exploit the GPU's capabilities. Furthermore, even if each GPU had a more favorable number of columns, there would still be 16 CPUs that must manage shared access to four GPUs, requiring either extra synchronization or communication to send data between CPUs with and without control of a GPU. Again, these problems are all solvable, but it might be difficult to shoehorn support for this type of system into an existing application that wasn't designed from the ground up to support it.

Clusters aren't going away, they're just becoming more complicated.

The limitations I've discussed here—Amdahl's law, programming language and porting difficulties for large models, the need for better programming tools, and the unsuitability of certain methods in the current models—would make it very difficult to port a large model such as CAM to a GPU one routine at a time. Ultimately, we don't see an inherent barrier to using GPUs to accelerate atmospheric models. We just don't think that porting existing models piece by piece is the best path forward.

The first full atmospheric model to truly use GPUs efficiently will likely need to be designed for a GPU from the ground up, with its data structures and numerical methods selected to suit the GPU's unique architectural features. Currently, there's a considerable convergence between the GPU world and the microprocessor world. CPUs are trading single-threaded performance for increasing core counts; at the same time, GPUs are moving away from their roots as highly task-specific architectures toward more flexible designs that better suit general computation. Ultimately, CPUs and GPUs will probably share a home on the same die. As their architectures evolve alongside each other, we hope the convergence will encourage the development of new tools and techniques that will benefit both sides—and allow atmospheric modeling to benefit from the computational power that GPUs promise.

References

1. D. Göddeke et al., "GPU Acceleration of an Unmodified Parallel Finite Element Navier-Stokes Solver," W.W. Smari and J.P. McIntire, eds., *High Performance Computing & Simulation*, Logos Verlag, 2009, pp. 12–21.
2. J.C. Thibault and I. Senocak, "CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows," *Proc. 47th AIAA Aerospace Sciences Meeting*, Am. Inst. Aeronautics and Astronautics, 2009, paper no. AIAA 2009-758.
3. V. Simek et al., "Towards Accelerated Computation of Atmospheric Equations Using CUDA," *Proc. 11th Int'l Conf. Modelling and Simulation*, IEEE CS Press, 2009; pp. 449–454; http://doi.ieeecomputersociety.org/10.1109/UKSIM.2009.25.
4. J. Michalakes and M. Vachharajani, "GPU Acceleration of Numerical Weather Prediction," *Parallel Processing Letters*, vol. 18, no. 4, 2008, pp. 531–548.
5. W.D. Collins et al., "The Formulation and Atmospheric Simulation of the Community Atmosphere Model Version 3 (CAM3)," *J. Climate*, vol. 19, no. 11, 2006, pp. 2144–2161.
6. W.D. Collins et al., "The Community Climate System Model Version 3 (CCSM3)," *J. Climate*, vol. 19, no. 11, 2006, pp. 2122–2143.
7. B.P. Briegleb, "Delta-Eddington Approximation for Solar Radiation in the NCAR Community Climate Model," *J. Geophysical Research*, vol. 97, no. D7, 1992, pp. 7603–7612.

***Rory Kelly** is a software engineer in the Computational and Information Systems Laboratory of the National Center for Atmospheric Research in Boulder, Colorado. His research interests include parallel and distributed computing, and using computational accelerators (GPUs, FPGAs, and so on) for scientific modeling. Kelly has a BA in physics and a BA in mathematics from the University of Colorado at Boulder. Contact him at rory@ucar.edu.*

RUNNING IN CIRCLES LOOKING FOR A GREAT COMPUTER JOB OR HIRE?

The IEEE Computer Society Career Center is the best niche employment source for computer science and engineering jobs, with hundreds of jobs viewed by thousands of the finest scientists each month - **in *Computer* magazine and/or online!**

- > Software Engineer
- > Member of Technical Staff
- > Computer Scientist
- > Dean/Professor/Instructor
- > Postdoctoral Researcher
- > Design Engineer
- > Consultant

careers.computer.org
http://careers.computer.org

The IEEE Computer Society Career Center is part of the *Physics Today* Career Network, a niche job board network for the physical sciences and engineering disciplines. Jobs and resumes are shared with four partner job boards - *Physics Today* Jobs and the American Association of Physics Teachers (AAPT), American Physical Society (APS), and AVS: Science and Technology of Materials, Interfaces, and Processing Career Centers.

IEEE computer society