High-performance Computation and Visualization of Plasma Turbulence on Graphics Processors

George Stantchev, Derek Juba, William Dorland, and Amitabh Varshney

***Abstract*—Direct numerical simulation (DNS) of turbulence is computationally very intensive and typically relies on some form of parallel processing. Spectral kernels used for spatial discretization are a common computational bottleneck on distributed memory architectures. One way to increase the efficiency of DNS algorithms is to parallelize spectral kernels using tightly-coupled Single-Program-Multiple-Data (SPMD) multiprocessor units with minimal inter-processor communication latency. We present techniques to map DNS computations to modern Graphics Processing Units (GPUs), which are characterized by a very high memory bandwidth and hundreds of SPMD processors. We compare and contrast the performance of our parallel algorithm running on a GPU versus the associated CPU implementation of a solver for one of the fundamental nonlinear models of turbulence theory. We also demonstrate a prototype of a scalable computational steering framework based on turbulence simulation and visualization coupling on the GPU.**

***Index Terms*—GPU Programming, Scientific Computing, Numerical Simulations, Plasma Turbulence**

I. Introduction

MAGNETIZED plasma (ionized gas) is the main fuel component in controlled thermonuclear fusion devices. Turbulence in plasma can lead to energy losses and various catastrophic events. It is therefore highly beneficial to develop a predictive understanding of plasma turbulence via high-fidelity numerical simulations. Currently such simulations are typically carried out on large parallel supercomputers, the access to which is generally limited, and whose cost of operation is high.

Inspired by the latest trends and developments in graphics processing technology we propose a new paradigm for implementing some of the major aspects of the plasma turbulence program. We argue that physically meaningful turbulence problems can be mapped efficiently to modern graphics processors, dramatically reducing cost while increasing both accessibility and performance In fact, in our preliminary investigations we have observed that simulation codes that can be run entirely on the graphics processor unit outperform considerably their CPU counterparts. We believe that bringing plasma turbulence simulations to the desktop PC will empower the current generation of plasma physicists with a computational tool that could increase significantly the productivity of their research.

This paper is organized as follows: in Section II we give an overview of some current trends in graphics processing architectures. Section III discusses the role of spectral methods in DNS of turbulence. In Sections IV and V we outline the theoretical aspects of the Hasegawa-Mima fluid turbulence model and its implementation on the graphics processor. Section VI discusses the performance of our Hasegawa-Mima solver relative to its conventional CPU implementation.

II. Overview of GPU computing

Graphics processors have always been characterized by parallelism, pipelining, large memories, and high bandwidths. Over the years, the graphics architecture has migrated from mainframes to workstations to PC cards, and now to Graphics Processing Units (GPUs). Modern GPUs exhibit two kinds of parallelism – functional parallelism (by pipelining) and data parallelism. The GPUs have sustained a super-Moore's law rate of growth for over a decade now [14]. The primary drivers of this phenomenal growth have been the highly parallel nature of graphics processing predicated on paradigms suchs as order-independence, pipelining, and streaming.

As a reference, the performance of the top NVIDIA G80 model is over 375 GFLOPS (about $1.50/GFLOPS) whereas the performance of a 3.0 GHz Intel Core2 Duo CPU is about 50 GFLOPS (about $4/GFLOPS) [1]. Note that this improved performance is achievable only if the problem maps well to the underlying processor architecture.

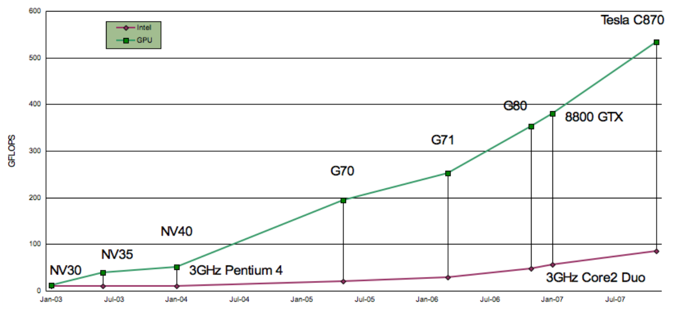
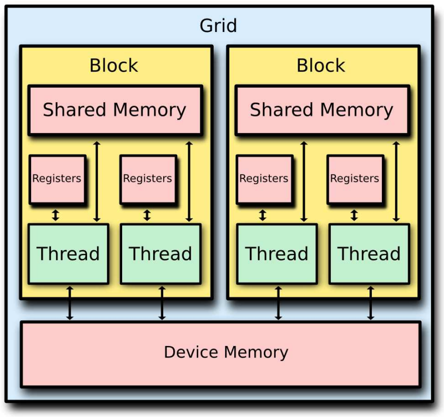


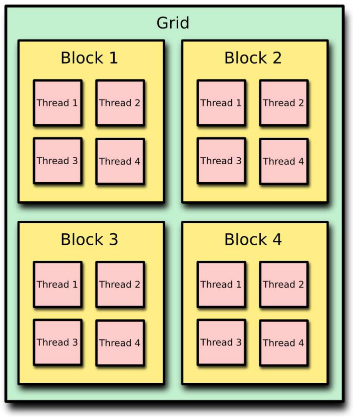
Fig. 1. Performance Comparison Among NVIDIA GPUs and Intel CPUs

The diagram on Fig. 1 is based on data compiled from [14], [1], and shows the rapid rise in the floating-point performance of NVIDIA GPUs as compared to Intel CPUs. GPU memory size is also growing, albeit more slowly, with current generation GPUs offering up to 1.5 GB of RAM. Current GPUs support only single-precision floating point arithmetic, but double precision support is expected on NVIDIA GPUs by the second half of 2008

On the latest generation of NVIDIA GPUs, the G8x series, the vertex and fragment processors from previous generations have been unified, and models with up to 128 such unified processors are available. The G8x also provides fast shared memory that can be used for inter-processor communication. Perhaps the biggest benefit of the new G8x GPU however, is the introduction of NVIDIA's Compute Unified Device Architecture (CUDA) [1]. Rather than using a standard graphics driver, CUDA accesses the hardware through a special driver designed for general purpose computing. CUDA also allows GPU programs to be written in ANSI C (with a few extensions), rather than languages like Cg or GLSL that were designed for shading algorithms. Finally, CUDA provides BLAS and FFT libraries intended to take advantage of the hardware's new capabilities. The GPU programming model is an approximation of the streaming-model of data computation. The ideal computational problem for such a model is one in which a small program (the kernel) operates upon an input stream of data to generate an output stream. The GPUs also permit parallel scatter and gather operations over the processors through shared memory (Fig. 2(a))



(a) Memory Model



(b) Thread-Block Hierarchy

Fig. 2. Memory and Programming Model Diagrams of NVIDIA's G8x GPU

The programming model of the CUDA architecture turns the NVIDIA G8x into a computing device equivalent to a highly multithreaded coprocessor. The parallelism of CUDA is exposed through a hierarchical execution grid structure consisting logically of threads and blocks (Fig. 2(b)). The indexing scheme supported by CUDA features up to three-dimensional indexing of threads within each block and independent two-dimensional indexing of blocks within the grid. Typically, when a kernel program is executed on the device, a given thread operates on a single element of each input/output array. The CUDA indexing scheme allows for easy one-to-one association between threads and individual elements of multi-dimensional arrays.

III. FFT: the Basic Computational Kernel of Turbulence Codes

A fundamental challenge in the simulation of turbulent dynamical systems is that relevant features need to be resolved over a wide range of length scales. Spectral methods provide an efficient way to deal with this issue without sacrificing accuracy. The basic tool used in spectral methods is the Discrete Fourier Transform (DFT) which translates the original partial differential equations into a system of algebraic equations in wave-vector space. Unlike Finite Difference/Finite Element schemes, spectral methods are global in that the evaluation of a quantity at a given computational grid node requires access to information at all other nodes. For instance, suppose $\{\phi(n)\}, n = 0, 1, \ldots, N$, are samples of a continuous function ϕ over a finite interval. In the one-dimensional case the Discrete Fourier Transform maps the vector $(\phi(0), \ldots, \phi(N-1))$ to a vector $(\widehat{\phi}(0), \ldots, \widehat{\phi}(N-1))$, such that:

$$\widehat{\phi}(k) = \sum_{n=0}^{N} \phi(n) e^{\frac{\pi k n}{N}}$$

The Inverse DFT is given by

$$\phi(n) = \sum_{n=0}^{N} \widehat{\phi}(k) e^{-\frac{\pi k n}{N}}$$

Evaluation of the derivative $\partial/\partial x$ in wave-vector space translates simply into multiplication by k. However, to obtain the corresponding value in physical space the Inverse DFT needs to be applied: this clearly requires access to the entire vector $\{\widehat{\phi}(n)\}$.

The global nature of spectral methods inherent in the DFT posits a challenge in parallelizing simulation codes that rely on such methods. Traditional domain partitioning techniques benefit from algorithms where most of the information is computed locally; global communication is needed only along partition boundaries. For spectral codes, several parallelization strategies exist: for instance, use a massively parallel vector processing architecture with Remote Memory Access functionality, such as the Earth Simulator [18]; this has the advantage of performing spectral discretization over the entire domain; the obvious disadvantages are access and cost. A more balanced approach is to use algorithms with spectral discretization along a subspace, and a local discretization scheme (usually finite differencing) along the complement. The applicability of such algorithms is dictated by the physical nature of the problem. In plasma turbulence, for example, there are regimes in which small scale turbulent structures develop in subspaces perpendicular to magnetic field lines, whereas along field lines characteristic scale lengths are comparatively long and thus do not require high spatial resolution in simulations [9], [3].

We focus on DNS methods for turbulence problems which admit such dimensional splitting. Suppose we have a 3D domain such that spectral discretization is carried out along planes P_z perpendicular to the z-axis. An optimal parallelization scheme would map a collection of P_z's to a single processor, thus ensuring that each DFT will be performed locally. In this sense the DFT on a rectangular box in each P_z can be considered as the *minimal parallelization unit (MPU)* of the implementation: further domain partitioning within P_z would be inefficient. The performance bottleneck of a simulation code in this scenario is determined by the speed of computing each MPU, which in turn depends on the performance characteristics of the underlying CPU architecture.

A typical paradigm for distributed memory architectures is to compute each MPU on a single processor in serial fashion using some implementation of the Fast Fourier Transform (FFT) algorithm. But the 2D FFT is a tensor product operation: it is carried out as a sequence of one-dimensional FFT's successively along the columns and the rows of the input array. Performing multiple one-dimensional FFT's simultaneously has the potential for achieving further data-parallelism and therefore bigger code acceleration. It is thus natural to map the computation of each 2D FFT to the Graphics Processing Unit (GPU) whose tightly-coupled multiprocessor SPMD architecture and high memory bandwidth facilitates the extra parallelization step with virtually no communication overhead.

The CUDA software development environment comes equipped with a high-performance FFT library, which is optimized for the specific architecture. It provides a simple interface for computing FFT's efficiently by leveraging the floating-point power and parallelism of the GPU. It supports batch execution for performing multiple 1D transforms in parallel and arbitrary array sizes up to 16384 per batch (not restricted to powers of 2). We have tested the performance of the CUFFT library in comparison with one of the most widely used CPU implementations of the FFT, the Fastest Fourier Transform in the West [5]. Figure (3) shows that even for relatively small array sizes speed gain can be considerable.

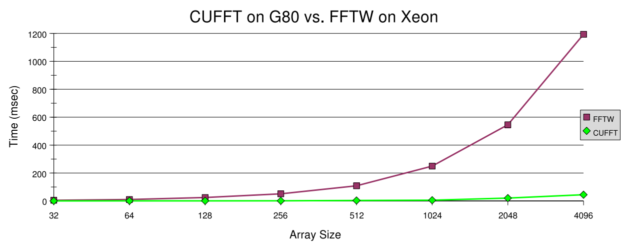


Fig. 3. CUFFT vs. FFTW on batches of parallel 1d FFT. Array size refers to the size of each batch; the number of batches run in parallel is set to 2048. The CPU is a 3GHz 64-bit Xeon

IV. The Hasegawa-Mima Equation

The quintessential nonlinear model for plasma turbulence theory for over three decades has been the Hasegawa-Mima model: [8]

$$\frac{\partial(1-\nabla_\perp^2)\Phi}{\partial t}+v_*\frac{\partial\Phi}{\partial y}-\mathbf{v}_E\cdot\nabla\nabla_\perp^2\Phi=0 \qquad (1)$$

In this equation, $\mathbf{v}_E \equiv c/B_0(\hat{\mathbf{b}}\times\nabla\Phi)$, v_* is proportional to the gradient of density in the plasma configuration, and Φ is the electrostatic potential. In general magnetic field-line-following coordinates, it is possible to express the nonlinear term as

$$-\mathbf{v}_E\cdot(\nabla_\perp^2\Phi)\equiv-\{\Phi,\nabla_\perp^2\Phi\},$$

where $\{f,g\}$ denotes the Poisson bracket, whose derivatives are evaluated in the plane locally perpendicular to the magnetic field. In Cartesian coordinates with the magnetic field in the $z-$direction, this would be $\{f,g\}=f_xg_y-f_yg_x$.

Physically, the Hasegawa-Mima equation describes the perpendicular motions of incompressible plasma turbulence in a strong magnetic field. The dynamics along the field line evolve separately from the dynamics perpendicular to the field line. The only nonlinearities are of the Poisson bracket type, in the perpendicular plane. Numerous models of magnetized plasma dynamics generally share these properties. The Hasegawa-Mima model is the simplest, as it involves a single field $\Phi(x,y)$ on a two-dimensional, periodic domain. More sophisticated models, such as Reduced MHD (RMHD), involve multiple fields and include parallel (along the field line) dynamics [12]. The simulation algorithm for the RMHD and other models could be very similar to the one used for the Hasegawa-Mima equation. Typically the difference is in the number of Poisson brackets, and consequently Fourier Transforms, that need to be evaluated at each time step. This number can be thought of as an estimate of the "computational intensity" of a particular algorithm. To the extent that this intensity measure maps well into the capabilities of the computing device and reflects the relative physical realism of the given simulation model, it is natural to expect very high performance from the GPU architecture for plasma turbulence simulations.

V. CUDA Implementation of the Hasegawa-Mima Equation Solver

For our experiments we started with our own CPU implementation of a Hasegawa-Mima (HM) solver written in Fortran 90. We had a choice of using the Fortran code as a control module that makes calls to CUDA kernel wrappers for all computationally intensive array operations. This approach was useful for initial testing but it became clear that in order to achieve optimal efficiency we must reduce data transfer to and from the GPU to a minimum. Even with the current fast PCI-Express bus, read/write operations are slow compared to memory transfer within the GPU. Thus we converted the majority of the Fortran code into C/CUDA and we left only data initialization and output diagnostics on the Fortran side. Once data has been passed to GPU memory it stays in until the end of the simulation.

The HM solver follows a classical time-stepping algorithm based on the 4th order Runge-Kutta Method. The most computationally intensive operation performed within a time-stepping cycle is the FFT. Our implementation of the HM solver requires 20 FFT's per time step: 16 complex-to-real and 4 real-to-complex. Arrays are kept in reduced complex format; due to its inherent Hermitian symmetry, the FFT image of a real array of size $N_1\times N_2$ can be represented by a complex array of size $N_1 \times (N_2/2+1)$. For grid sizes larger than 1024x1024 grid points, this type of reduction is essential in order to avoid the 768MB memory limitation of most G8x cards. All other operations required in the time-stepping loop are linear- or sublinear-in-time pointwise vector arithmetic operations.

Figures (4) and (5) illustrate the basic steps in our HM solver implementation.

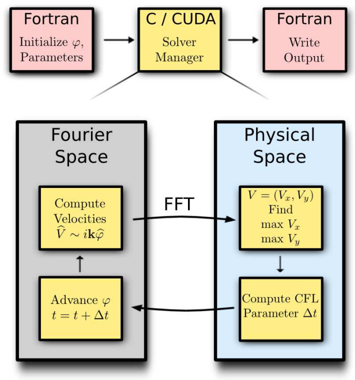

Fortran
Initialize φ, Parameters
C / CUDA
Solver Manager
Fortran
Write Output
Fourier Space
Physical Space
Compute Velocities $\widehat{V} \sim i\mathbf{k}\widehat{\varphi}$
FFT
$V=(V_x,V_y)$ Find max V_x max V_y
Advance φ $t=t+\Delta t$
Compute CFL Parameter Δt

Fig. 4. Overview of the HM Solver on the GPU

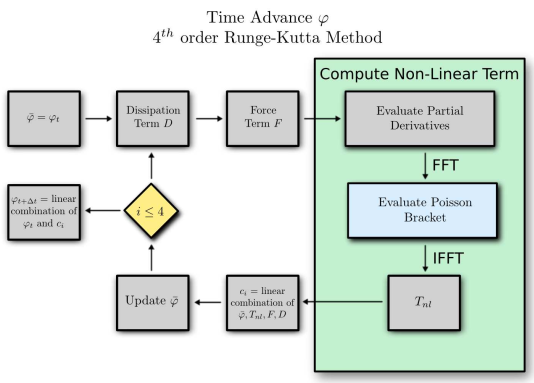

Time Advance φ
4^{th} order Runge-Kutta Method
Compute Non-Linear Term
$\bar{\varphi}=\varphi_t$
Dissipation Term D
Force Term F
Evaluate Partial Derivatives
FFT
Evaluate Poisson Bracket
IFFT
T_{nl}
$\varphi_{t+\Delta t}$ = linear combination of φ_t and c_i
$i \leq 4$
Update $\bar{\varphi}$
c_i = linear combination of $\bar{\varphi}, T_{nl}, F, D$

Fig. 5. The Time-stepping Algorithm

A. Visualization and Computational Steering

Since data is already available on the GPU, it is natural to visualize the results of the turbulence simulation directly from the graphics card. This is important since turbulence simulation diagnostics keep track not of individual scalar field values but of derived statistical quantities, typically a few numbers per time frame, thus obviating the need to transfer significant amounts of data back to CPU memory.

Data produced by the simulation is processed for immediate rendering at each time step. The visualization follows a familiar pipeline starting with a GPU kernel function to perform color mapping of data values. The resulting buffer of RGB color values is then bound as an OpenGL buffer object using CUDA's OpenGL interoperability functions. This buffer of pixel data is made available for rendering either by binding it as a texture map and then rendering a textured polygon, or by drawing its pixels directly to the screen. Figure (6) shows the electric potential field at the onset of turbulence in a Hasegawa-Mima simulation at resolution 512×512.

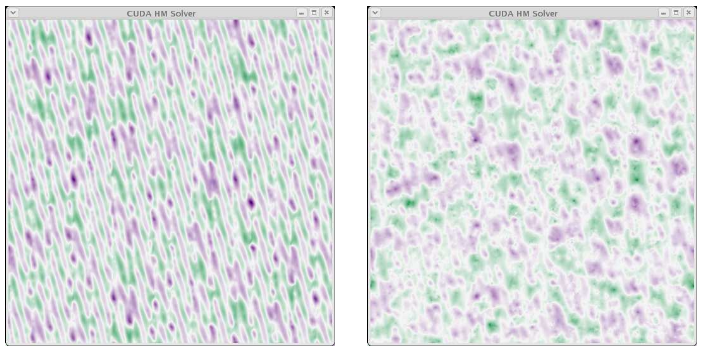


Fig. 6. Concurrent Visualization: Onset and Development of Turbulence

In addition to visualizing the simulation as it runs, simulation parameters can also be interactively adjusted. We call the GPU simulation kernel from a CPU driver function that repeatedly makes calls to the kernel in a loop to advance the current time step. This loop also checks for user input events, and can modify parameters correspondingly for the next kernel call.

VI. Performance Results

We have evaluated the performance of our CUDA implementation of the Hasegawa-Mima equation solver versus the corresponding Fortran implementation running on a 3.0GHz 64-bit Intel Xeon processor. We measured the wall clock time for executing 100 steps of the simulation's main loop with identical setup parameters. For small input array sizes the CUDA implementation is relatively slower due to overhead in kernel call initialization latency. At array sizes larger than 128×128, however, we observed increasingly bigger speedup factor, with maximum of 14 at resolution 1024×1024. Results from our comparison experiments are presented in Fig. 7. It is important to note that bigger relative speedup can be achieved if all Fourier Transforms are performed in full complex-to-complex format, at the expense of doubling the simulation's memory footprint and increasing the absolute execution time.

We have also evaluated the performance of a CUDA implementation for a numerical solver of the Reduced MHD equations [16]. In this case we observe a speedup factor of 25-30, depending on the size of the computational grid (see Fig. 8). This increase in CUDA-to-CPU speedup compared to the Hasegawa-Mima solver is mainly due to the larger number of nonlinear terms present in the RMHD equations (three vs one for Hasegawa-Mima). Since each nonlinear term translates computationally into a set of Fourier Transforms, more nonlinear terms means a larger portion of the simulation is spent calculating FFTs, i.e. on operations for which CUDA's performance gain over the CPU is much higher compared to those used elsewhere in the code (see also discussion in Section VII).

VII. Conclusion

We have demonstrated that carrying out DNS of plasma turbulence on the GPU provides a significant performance benefit in comparison with the CPU; also, implementation

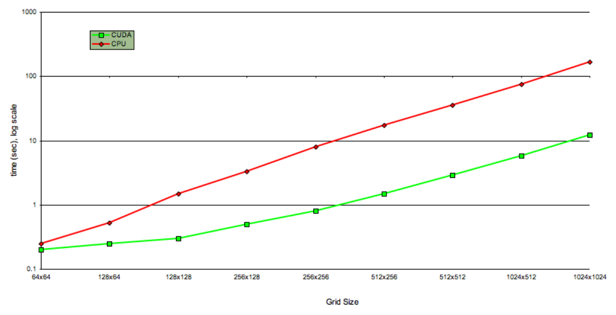


Fig. 7. CUDA/CPU performance comparison for the Hasegawa-Mima solver. On the vertical axis is wall clock time for the execution of 100 steps of the main loop

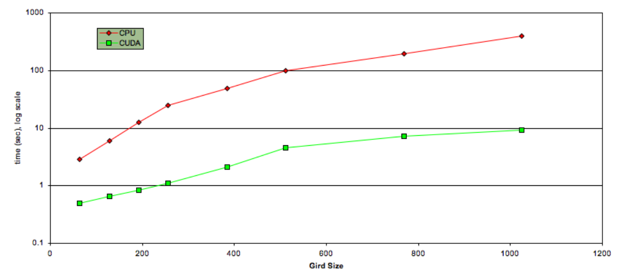


Fig. 8. CUDA/CPU Performance Comparison for the RMHD Solver. The horizontal axis represents the size N of one of the grid's two dimensions. The total grid size is $N \times N$

is greatly facilitated by the latest CUDA programming environment on NVIDIA's G8x cards. Due to the considerable overhead of data transfers to and from the graphics card, the major limitation is GPU memory size. However, many physically meaningful simulations fit within the current GPU memory limits In particular, for many problems of interest in fusion energy research, spectral resolution of 1024×1024 can be fairly sufficient [3], [10]; usually of bigger concern is the inclusion of extra non-linear terms in the modeling equation. These non-linear terms correspond to interactions among physical quantities tracked by the model and thus contribute to the simulation of various important physical effects; mathematically they can be expressed in the form of Poisson brackets, which in turn translate into extra FFT's per time step. This increase in the relative dominance of computational complexity over memory requirements is one of the major leveraging factors in favor of implementing spectral DNS methods on the GPU. In summary, for a fixed spectral resolution *the more physics is added to the model, the bigger the expected speedup factor over the CPU.*

As future work we envision a thorough performance analysis of the GPU implementation of a general class of spectral methods for direct numerical simulation of turbulence. In particular, it is important to quantify the average speedup factor per non-linear term at various resolutions.

We also plan to study the implementation of a different class of methods, namely those based on particle-in-cell simulations [15]. These methods require tracking of a large number of virtual particles whose positions and velocities are updated at each time step according to a system of ordinary differential equations. We expect that even in the absence of bottlenecks of "hyper-linear" computational complexity such as the FFT, a GPU particle code implementation will still have a measurable speedup factor over the CPU. This is due to the fact that particles are advected independently of one another and thus can be partitioned in any arbitrary fashion; this allows for optimal kernel parameter configuration and hence should lead to achieving maximum performance. In contrast with spectral methods, however, particle resolution and hence memory complexity has larger relative significance: the number of particles strongly affects the accuracy of the simulation. One approach to this problem is to scale the implementation to a GPU-enabled PC cluster using standard domain partitioning techniques. A GPU cluster based on previous generation of NVIDIA cards has already been used successfully in applying a parallel Lattice Boltzmann Method to several problems in computational fluid dynamics [17]. We plan to investigate the efficiency of mapping particle-in-cell algorithms for plasma dynamics to scalable high-performance distributed GPU architectures.

Concurrent visualization is another direction of research which is important to pursue. Moving from two- to multi-dimensional data is likely to put non-trivial burden on the GPU especially if computationally intensive techniques such as volumetric rendering are being used. In order to maintain user interactivity it will become expedient to rely on multi-GPU nodes where several GPUs would typically run the simulation and one would handle visualization. Dealing with synchronization and load balancing would become of critical importance. We plan to investigate the viability of implementing a scalable computational steering framework based on GPU technology in the context of DNS of plasma turbulence.

Acknowledgments

The authors would like to thank Dr. David Luebke for his support and assistance with the CUDA programming environment. We are grateful Dr. Tomoya Tatsuno and Kyle Gustafson for helpful discussions on the physics of plasma turbulence simulations. We thank Prof. Ramani Duraiswami and Dr. Nail Gumerov for sharing their experience with CUDA scientific computing. We also appreciate all comments and suggestions made by the paper referees.

This work was partially supported by NSF grant # CISE RI 04-03313.

References

[1] CUDA Developer's Zone. http://www.nvidia.com/object/cuda_home.html, 2008.

[2] Ian Buck. Gpu computation strategies & tricks. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 134, New York, NY, USA, 2005. ACM Press.

[3] W. Dorland, F. Jenko, M. Kotschenreuther, and B. Rogers. Electron temperature gradient turbulence. *Phys. Rev. Lett.*, 85:5579, 2000.

[4] D. H. E. Dubin, J. A. Krommes, C. Oberman, and W. W. Lee. Nonlinear gyrokinetic equations. *Physics of Fluids*, 26:3524, 1983.

[5] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on ”Program Generation, Optimization, and Platform Adaptation”.

[6] Naga Govindaraju and Dinesh Manocha. GPUFFTW: High performance power-of-two FFT library using graphics processors. http://gamma.cs.unc.edu/GPUFFTW/index.html, 2005.

[7] N. Gumerov, R. Duraiswami, and W. Dorland. Middleware for programming NVIDIA GPUs from Fortran 9x, 2007. Supercomputing ’07, Reno, NV.

[8] A. Hasegawa and K. Mima. Stationary spectrum of strong turbulence in magnetized nonuniform plasma. *Physics Review Letters*, 39:205, 1977.

[9] G. G. Howes, S. C. Cowley, W. Dorland, G. W. Hammett, E. Quataert, and A. A. Schekochihin. Astrophysical gyrokinetics: Basic equations and linear theory. *Astrophysics Journal*, 651:590, 2006.

[10] M. Kotschenreuther, W. Dorland, G. W. Hammett, and M. A. Beer. Quantitative predictions of tokamak energy confinement from first-principles simulations with kinetic effects. *Physics of Plasmas*, 2:2381, 1995.

[11] Kenneth Moreland and Edward Angel. The FFT on a gpu. In *HWWS ’03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[12] P. J. Morrison and R. D. Hazeltine. *Hamiltonian formulation of reduced magnetohydrodynamics*. Unknown, July 1983.

[13] W. M. Nevins, G. W. Hammett, A. M. Dimits, W. Dorland, and D. E. Shumaker. Discrete particle noise in particle-in-cell simulations of plasma microturbulence. *Physics of Plasmas*, 12:122305, 2005.

[14] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[15] S. E. Parker and W. W. Lee. A fully nonlinear characteristic method for gyrokinetic simulation. *Phys. Fluids B*, 5:77, 1992.

[16] A. A. Schekochihin, S. C. Cowley, W. Dorland, G. W. Hammett, G. G. Howes, E. Quataert, and T. Tatsuno. Kinetic and fluid turbulent cascades in magnetized weakly collisional astrophysical plasmas. http://arxiv.org/abs/0704.0044, 2007.

[17] Xiaoming Wei, Wei Li, Klaus Mueller, and Arie E. Kaufman. The lattice-boltzmann method for simulating gaseous phenomena. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):164–176, 2004.

[18] Mitsuo Yokokawa, Ken’ichi Itakura, Atsuya Uno, Takashi Ishihara, and Yukio Kaneda. 16.4-tflops direct numerical simulation of turbulence by a fourier spectral method on the earth simulator. In *Supercomputing ’02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.