Accelerating geoscience and engineering system simulations on graphics hardware

Stuart D.C. Walsh a,*, Martin O. Saar a, Peter Bailey b, David J. Lilja b

a University of Minnesota, Department of Geology and Geophysics, 310 Pillsbury Drive S.E., Minneapolis, MN 55455-0219, USA
b University of Minnesota, Department of Electrical and Computer Engineering, USA

A R T I C L E   I N F O
Article history:
Received 26 September 2008
Received in revised form 17 April 2009
Accepted 14 May 2009

Keywords:
General purpose graphics processing units
Accelerators
Parallel computing
Lattice Boltzmann
Spectral finite element method
Least-squares minimization
Geofluids
Seismology
Magnetic force microscopy

A B S T R A C T
Many complex natural systems studied in the geosciences are characterized by simple local-scale interactions that result in complex emergent behavior. Simulations of these systems, often implemented in parallel using standard central processing unit (CPU) clusters, may be better suited to parallel processing environments with large numbers of simple processors. Such an environment is found in graphics processing units (GPUs) on graphics cards.

This paper discusses GPU implementations of three example applications from computational fluid dynamics, seismic wave propagation, and rock magnetism. These candidate applications involve important numerical modeling techniques, widely employed in physical system simulations, that are themselves examples of distinct computing classes identified as fundamental to scientific and engineering computing. The presented numerical methods (and respective computing classes they belong to) are: (1) a lattice-Boltzmann code for geofluid dynamics (structured grid class); (2) a spectral-finite-element code for seismic wave propagation simulations (sparse linear algebra class); and (3) a least-squares minimization code for interpreting magnetic force microscopy data (dense linear algebra class). Significant performance increases (between 10× and 30× in most cases) are seen in all three applications, demonstrating the power of GPU implementations for these types of simulations and, more generally, their associated computing classes.

© 2009 Elsevier Ltd. All rights reserved.

1. Introduction

Many processes of interest in both science and engineering involve complex systems characterized by the interactions of smaller component elements. This is particularly true of many phenomena observed in the geosciences, which often span an enormous range of spatial (μm to Earth) and temporal (μs to million year) scales. Such scales are commonly encountered in studies of climate systems (Fournier et al., 2004), tsunamis (Schmidt et al., 2009), seismic wave transmissions (Tsuboi et al., 2005), magma flow in the Earth’s mantle or crust leading to volcanic eruptions (Gonnermann and Manga, 2007), and glacial melting (Jellinek et al., 2004), to name a few.

The need to understand the emergent behavior of these complex systems creates a constant and growing demand for more powerful computational algorithms and hardware capable of solving such physical modeling problems (e.g., the Earth Simulator, Tsuboi et al., 2005). Often, this demand is not due to the algorithmic complexity of the problem, as a hallmark of many complex systems is the simplicity of their constituent elements (Wolfram, 1983). Instead, the principal limitation is the need to repeat many simple tasks a large number of times. Traditionally, this is achieved by implementing the problem on a central processing unit (CPU) cluster. However, standard cluster CPUs are unnecessarily sophisticated for many physical simulations. Thus, the speed, size, and detail of these simulations could be dramatically increased by using clusters of significantly larger numbers of smaller, simpler, and thus cheaper processors.

Such processor clusters already exist. A graphics processing unit (GPU) is hardware that currently costs around $500. Individual GPUs may contain 128 or more simple processors, thus constituting parallel computing systems designed to perform many simple simultaneous calculations.

GPUs were first co-opted for general purpose computing by manipulating the graphics pipeline. In particular, the advent of shaders, small customized routines inserted at specific stages of the pipeline (for example the stages that dealt with vertex and fragment graphics primitives), permitted savvy users to accelerate certain applications by recasting them in terms of graphics manipulations (Lindholm et al., 2001; Purcell et al., 2002). Excellent overviews of the types of problems investigated using shaders can be found in Lastra et al. (2004) and Owens et al.
Shader implementations were often limited to reduced graphics operations to reproduce the desired computation (Owens et al., 2007). The need to cast algorithms in terms of graphics operations not only adds to the complexity of shader programming but also restricts the manner in which data are processed. This means, for example, that although the shader model is efficient at simple element-wise operations, more complex data interdependencies have to be evaluated through more costly “multipass loops” (Hillesland et al., 2003; Pieters et al., 2007). Programmers are also restricted by certain hardware limitations, e.g., shader instruction count and number of shader outputs (Buck et al., 2004). As a result, although GPU shaders were able to achieve impressive performance gains in some cases, the limitations and complexities of the programming model prevented widespread adoption of shader GPU computing for scientific and engineering applications.

More recently, however, compilers have been released that allow custom-made programs to run directly on GPUs without adopting the graphics pipeline approach employed by shaders (e.g., Nickolls et al., 2008). In particular, in this paper we discuss the Compute Unified Device Architecture (CUDA) programming model released by nVIDIA (2008a). Other examples of this new generation of programming tools for the GPU include BrookGPU (Buck et al., 2004) and the soon-to-be-released OpenCL.

CUDA is focused on providing general purpose GPU (GPGPU) functionality rather than graphics programming. The CUDA model provides a set of minimal extensions to the C programming language that allow the programmer to write kernels—functions executed in parallel on the GPU. This frees the programmer from having to cast the problem in terms of a graphics context and removes the cost of calling the graphics application programming interface. In addition to making GPGPU programming more accessible, CUDA also provides new functionality that distinguishes it from the shader approach. For example, CUDA includes random access byte-addressable memory, and threads that can read and write to as many locations as needed. It also supports coordination and communication among processes through thread synchronization and shared memory—thereby allowing complex data dependencies to be processed without requiring multipass loops. Finally, CUDA supports both single and double precision, and IEEE-compliant arithmetic (nVIDIA, 2008a).

Studies have demonstrated that GPGPU implementations can deliver performance gains of 10× to 100× compared to single CPU implementations (e.g., Jeong et al., 2006; Anderson et al., 2007; Tölke and Krafczyk, 2008). Further, because GPUs are low-cost, high-volume commodity products that are constantly improved for gaming/video purposes, it may reasonably be expected that the GPU-computing speed advantage will hold pace with, if not outstrip, CPU development (Owens et al., 2007). Hence, orders of magnitude in cost savings, coupled with orders of magnitude computation speedups could result in truly transformational computing opportunities in science/engineering that would continuously outpace what could be achieved with traditional CPU clusters.

The question becomes then, is GPU-based scientific computing a more promising approach than, for example, multi-core parallelization, thus could GPU computing potentially revolutionize the way computational science/engineering is conducted? Or, is the GPU approach merely a short-term phenomenon that is limited in scope and applicability? While several scientists have successfully employed GPU computation in fields ranging from astrophysics (Kaehler et al., 2006) to quantum mechanics (Anderson et al., 2007), currently lacking is a systematic analysis of massively parallel GPU-computing and its potential for scientific/engineering modeling as well as implementations of such a system on a larger scale for more complex systems. To discover what potential impact GPU computation may have on scientific and engineering computing in general, it is necessary to evaluate systematically the classes of numerical methods that can be effectively implemented on massively parallel GPU systems. Here, we take the first steps of such an evaluation by investigating the potential impact of GPU implementations on a selection of example problems from the geosciences.

The technical report “The Landscape of Parallel Computing Research: A View from Berkeley” by Asanovic et al. (2006) describes 13 fundamental computing classes, of which seven (dense linear algebra, sparse linear algebra, spectral methods, N-body methods, structured grids, unstructured grids, and Monte-Carlo methods), originally identified by Colella (2004), are believed to be important for science and engineering applications. Each class comprises related computations and data movement, hence the performance of one method is indicative of the performance of others in that class. Thus, the significance of GPU-based scientific and engineering computing may be evaluated by studying the applicability of GPU computations to the original seven fundamental scientific/engineering numerical method classes. If a significant subset of these seven classes lend themselves to GPU-computing, new transformational opportunities would emerge in scientific computing, as orders of magnitude more (simple but sufficient) processors could result in substantially faster, larger, or higher-resolution simulations.

This paper focuses on the applicability of GPU computing to problems in the geosciences. GPU computations appear well suited to geoscientific problems due to the spatial and temporal scales involved, as discussed at the beginning of this section. We have selected three applications for investigation, involving three of the seven computing classes deemed fundamental for scientific and engineering applications (Table 1):

1. a lattice-Boltzmann code for geofluid dynamics;
2. a spectral-finite-element code for seismic wave simulations; and
3. a least-squares minimization code for magnetic force microscopy data analysis.

The three programs involve important numerical modeling techniques routinely employed in a wide range of scientific/engineering disciplines beyond the geosciences.

A brief description of the GPU hardware used to run the simulations is given in Section 2. The lattice-Boltzmann simulation is described in Section 3, followed by the spectral-finite-element seismic-wave simulation in Section 4, and the least-cost

<table>
<thead>
<tr>
<th>Application</th>
<th>Numerical method</th>
<th>Computing class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational fluid dynamics</td>
<td>Lattice Boltzmann</td>
<td>Structured grid</td>
</tr>
<tr>
<td>Seismic wave propagation</td>
<td>Spectral finite element</td>
<td>Sparse linear algebra</td>
</tr>
<tr>
<td>Rock magnetism</td>
<td>Least squares minimization</td>
<td>Dense linear algebra</td>
</tr>
</tbody>
</table>
squares method in Section 5. At the beginning of each section, we provide an overview of the scientific computing applications that will be investigated and identify the fundamental numerical class to which each method belongs before discussing the example geoscience applications.

2. GPU hardware

The simulations conducted in this paper are performed using an nVIDIA (2008b) GeForce 8800 GTX graphics card, which has a peak performance of 518 GFLOPs and a memory bandwidth of 86 GBs. These graphics cards offer a particularly convenient platform for developing new GPGPU computing codes as they support code written using nVIDIA’s CUDA programming environment.

The internal structure of a GeForce 8 series GPU consists of up to 16 multiprocessors (nVIDIA, 2008a), specifically designed for compute-intensive, highly parallel computations. The GPU architecture provides a parallelism advantage over single- or dual-core general-purpose CPUs, which are capable of relatively few floating point operations per clock cycle. Computation on the GPU is organized into kernels, or GPU programs to be executed on some input data by multiple threads in parallel. All threads within a thread block execute the same kernel, communicate with each other through shared on-chip memory, and synchronize their computation with built-in synchronization instructions. Typically, multiple thread blocks are employed, because the hardware places constraints on the maximum number of threads in a single block. Thread blocks cannot synchronize executions as easily as threads within a single block can, nor do thread blocks link to on-chip memory. However, the blocks have access to global memory on the GPU card. This constraint limits thread-to-thread communication and puts a restriction on the amount of work that can be done in one kernel invocation.

Although the GPU architecture places constraints on inter-thread communication, it is specifically designed to optimize the throughput of a single set of instructions operating simultaneously on a large number of data sources. This single instruction, multiple data (SIMD) framework originally arose from the need for rapid processing of graphics data, in which the same simple graphics operations are performed on multiple pixels at once. Nevertheless, this framework is equally well suited to simulating the type of geophysical systems described in the Introduction, i.e., complex composite systems consisting of many simple interacting component parts.

3. Lattice-Boltzmann simulations—example application: geofluidic flows

Lattice-Boltzmann simulations are a method for modeling fluid mechanics in which the fluid is represented by a set of discrete fluid packets moving through a regular node lattice. These simulations are capable of modeling a wide range of complex fluid flow systems that are often intractable with other modeling methods—for example, complex and changing boundary geometries (Bosl et al., 1998; Hersum et al., 2005), turbulent and laminar flows (Chen et al., 1992), multiphase-multicomponent flow of miscible and immiscible fluids (Pan et al., 2004), and buoyancy-induced convection due to solute and thermal gradients (Alexander et al., 1993). These properties make lattice-Boltzmann methods particularly attractive for a large number of geofluid and other fluid-mechanical applications. Lattice-Boltzmann simulations model porous media flows (Bosl et al., 1998), ocean circulation (Salmon, 1999), quantum fluids (Boghosian and Taylor, 1998), granular flows (Herrmann et al., 1994) and colloidal suspensions (Ladd, 1993). The price of the relative ease of implementing lattice-Boltzmann methods is the large amount of CPU time these simulations demand. These costs can be offset in part through the use of non-uniform grids and mesh refinement methods (e.g., Filippova and Hanel, 1998; Lee and Lin, 2003; Li et al., 2005). Here, an alternative, but complementary approach, is explored, accelerating lattice-Boltzmann methods with GPU implementation.

Here, the ability of GPU implementations to accelerate lattice-Boltzmann simulations is highlighted by considering one particular application: determination of pumice permeability from fluid flow simulations through X-ray tomography images. Magmatic volatile degassing is typically impossible to measure directly within the volcano conduit, particularly during eruptions. Understanding the details of volcanic degassing, however, is widely considered critical in determining volcanic eruption dynamics, specifically the transition between effusive and explosive eruptions (Hammer et al., 1999). Although considerable effort has been made to investigate the complex flow and degassing behavior of magma (e.g., Saar et al., 2001; Saar and Manga, 2002; Jellinek et al., 2004; Walsh and Saar, 2008), the limitations of current computational tools place considerable constraints on the scale and type of problems that can be simulated, even when traditional supercomputing clusters are employed. For example, to determine volatile degassing rates at the appropriate volcanic conduit scale, eruption products such as pumice samples have to be investigated at a much smaller scale and then up-scaled. To do this, we use the Lawrence Berkeley National Laboratory Synchrotron facility to obtain three-dimensional tomography scans of cored pumice samples with resolutions of about 4 µm to capture the thin inter-bubble walls (Fig. 1a). Even before upsampling, the size of the three-dimensional tomography data precludes lattice-Boltzmann simulations of gas flow through the whole sample (one gigavoxel is equivalent to 0.064 ml at 4 µm resolution). A GPU approach has the potential to provide the computational means to solve these problems.

At each timestep in the lattice-Boltzmann model, fluid packets undergo a two-step process: (1) a streaming step in which the discrete fluid packets are propagated between neighboring nodes; and (2) a collision step in which the fluid packets converging on individual nodes are redistributed according to a set of simple rules (Wolf-Gladrow, 2000). For example, if the node is part of the fluid domain, then the fluid packets are updated according to

\[ f_i(t + \Delta t) = (1 - \alpha) f_i(t) + \alpha f_i^{eq}(t), \]

where \( f_i(t) \) is the fluid-packet density for the lattice velocity, \( i \), at time, \( t \), \( \alpha \) is a relaxation constant that determines the viscosity of the fluid and \( f_i^{eq}(t) \) is the local-equilibrium fluid packet density. The exact form of the equilibrium fluid packet densities is determined by the type of lattice being simulated. The GPU implementation discussed here is based on the popular D3Q19 lattice (i.e., a three-dimensional nineteen-speed lattice, Fig. 2a) where

\[ f_i^{eq} = \rho w_i \left( 1 - \frac{3}{2c^2} u \cdot u + \frac{c^2}{2} c_i \cdot u + \frac{9}{2c^2} (\mathbf{c} \cdot \mathbf{u})^2 \right), \]

in which \( w_i \) are lattice velocity-specific constants, \( \mathbf{c} \) are the lattice velocities, \( c \) is the lattice speed, \( \rho = \sum_i f_i(t) \) is the macroscopic fluid density and \( \mathbf{u} = \sum_i f_i(t) \mathbf{c}_i / \rho \) is the macroscopic fluid velocity (Qian et al., 1992). More detailed discussion on the lattice-Boltzmann method and implementation of other material models including boundary conditions is given in Succi (2001) and Ginzburg et al. (2008). Due to the ordered nature of the lattice and the predictable manner in which the fluid packets propagate,
Lattice-Boltzmann models are examples of the “Structured Grid” class (Table 1) described in the Berkeley report (Asanovic et al., 2006).

Lattice-Boltzmann simulations are particularly suited to GPU implementation. Before each simulation, the total-system CPU designates each node as either a fluid or a boundary node to reflect the simulation geometry. As a consequence of the simple nature of the rules governing the lattice-Boltzmann simulation, once the geometry is loaded, the remainder of the simulation occurs entirely on the GPUs. Further, CPU–GPU communication need only occur when GPUs return flow field “snapshots” to the system CPU.

Our GPU algorithm for the lattice Boltzmann program proceeds as follows:

1. Initial fluid packet densities and lattice geometry are loaded into GPU memory. The fluid packets are stored in a single one-dimensional array, arranged in order according to direction and then position (z then y then x) so that fluid packets propagating in the same direction are grouped together in the array.
2. One block of threads on the GPU is assigned to each row of lattice Boltzmann nodes along the x-axis, the threads within
At each simulation timestep:

1. Individual threads load the corresponding fluid packets arriving at their assigned node at that timestep. Fluid-packet propagation in the $y$- and $z$-axis directions is performed implicitly, by selectively loading the incoming fluid packets from different locations in the array of fluid packet densities (Fig. 4) into the shared memory for the thread block. Thus, the fluid packets are stored in global memory in a Lagrangian manner (i.e., in the same configuration as at the first timestep), while the thread block operates on the fluid packets in an Eulerian fashion (representing the change in the fluid packets at the node).

2. The individual threads perform the collision step for each node using the fluid packets stored in the thread block's shared memory.

3. Fluid packets with $x$-velocity components are propagated along the rows, by copying between locations in the thread block's shared memory. This rearrangement is needed to ensure that the fluid packets are written back to global memory in a coalesced fashion in the following step.

4. Outgoing fluid packets are written back to the original locations in GPU memory (Fig. 4).

(4) The implicit propagation will leave the fluid packets disordered following the final timestep, unless both $y$ and $z$ lattice dimensions are factors of the number of timesteps. Consequently, the final step of the algorithm reorders the fluid packets so their $y$ and $z$ locations correspond to the correct position in the lattice-Boltzmann array. The correction step in the $y$ direction is performed using thread blocks of the same size as before arranged in a two dimensional grid with dimensions $GCF_y \times N_y$, where $N_y$ is the number of nodes along the $z$ axis, and $GCF_y$ is the greatest common factor between $N_y$ (the number of nodes along the $Y$-axis) and the required offset in the $y$ direction, itself equal to the number of timesteps modulo, $N_y$. The correction is performed by each thread repeatedly shifting fluid packets with a $y$ velocity component forward to the correct position in the array, replacing the old value at that location which is then itself propagated forward to the next location in the fluid packet array. The same procedure is performed for the correction step in the $z$ direction.

Our GPU implementation is similar to those described in Tölke (2008) and Tölke and Krafczyk (2008), in that the key to the improved performance of all three simulations is the row-wise propagation of the fluid packets in Step 3(c). This row-wise propagation ensures coalesced memory access from the global memory in first generation CUDA devices. It should be noted, however, that this detail may be obsolete in the future, as the criteria for memory coalescence have been relaxed in later-generation CUDA devices (those with compute capability 1.2 and higher).

In particular, it may be that Step 3(c) could be replaced by implicit streaming methods in all directions as performed in Step 3(a). Here, the use of implicit streaming steps in $y$- and $z$-axis directions allows the fluid packets to be stored in a single array (Fig. 4), rather than separate arrays for incoming and outgoing fluid packets as in Tölke (2008) and Tölke and Krafczyk (2008). This reduces the total memory required to store the lattice, increasing the maximum number of nodes per GPU by almost a factor of two.

The GPU implementation of the lattice-Boltzmann model delivers substantial performance gains over the standard CPU implementation. For the D3Q19 lattice, we are able to achieve over 200 million lattice node updates per second (MLU/s). This figure is largely independent of lattice size (Table 2), although it is contingent on the entire lattice being stored in the GPU memory. This assumption is not unreasonable for the purpose of comparing the performance of a single CPU to a GPU–GPU system. While most GPU memories are smaller than those of their host CPU, this is not true for all GPUs—particularly those designed for general purpose computing rather than graphics operations. The nVIDIA C1060, for example, has 4 GB of onboard memory. Moreover,
multiple GPUs may be connected to a single CPU. If the amount of host CPU memory is less than, or equal to, the total GPU memory, then there is no benefit to communication between the CPU and the GPU, unless the simulation is larger than can be contained in a single CPU–GPU system.

Even greater performance gains in individual GPUs are reported by Tölke and Krafczyk (2008), who achieve speeds of up to 582 MLU/s using a multiple relaxation time D3Q13 lattice (i.e., the three-dimensional, thirteen-velocity lattice described by d’Humières et al., 2001). Part of the difference in performance is accounted for by the type of GPU employed—the simulations in Tölke and Krafczyk (2008) were conducted using a GeForce 8800 Ultra, which has a faster clock speed and greater bandwidth than the GeForce 8800 GTX used in this paper. However, we attribute the majority of the speed difference to the different number of lattice velocities (13 versus 19), which impacts performance in two ways: (1) the 19-velocity lattice requires a greater number of global memory reads and writes per node—a common bottleneck in GPU codes, (2) the reduced number of fluid packets allows the D3Q13 code to achieve greater occupancy (i.e., more efficient use) of the processors, and is thereby capable of executing a greater number of simultaneous instructions (Bailey et al., 2009).

However, despite the potential for an additional 2.5× speedup afforded by the more sparsely connected D3Q13 lattice, there are advantages to D3Q19 simulations—in particular for porous flow simulations based on voxelized tomography images, like those considered in this section. When simulating flow through a three-dimensional Cartesian grid, the D3Q13 lattice may be decomposed into two independent lattices that intertwine in a checkerboard fashion (Fig. 2b). These two independent lattices are problematic for porous medium flows as the effective resolution of the flow domain is reduced. Alternatively, the D3Q13 model can be employed in a single rhombic-dodecahedron lattice; however, this requires remeshing of voxelized data, which may also result in some loss of resolution. In contrast, all nodes in the D3Q19 lattice are interconnected (Fig. 2a), offering greater resolution of small scale apertures.

Figs. 1b and c show simulated fluid flow through a numerical core taken from a tomography image of a pumice sample. The core has dimensions of 128 × 128 × 512 voxels, corresponding to sample dimensions of 0.5 mm × 0.5 mm × 2 mm. Approximately 100,000 lattice-Boltzmann timesteps were run for one and a half hours. In contrast, running the same simulation on the CPU alone requires approximately two days computation time. The speed increase provided by the GPU allows multiple cores (e.g., with long axis aligned in the x, y and z directions) to be investigated for several different pumice samples in the same time that it would take to simulate a single sample on the same CPU without the GPU. In a broader context, the speedup provided by the lattice-Boltzmann GPU implementation enables thorough parameter-space investigations to be conducted for any number of different geoscience or engineering problems in circumstances where previously only a handful of simulations would have been possible.

### 4. Spectral-finite-element method—example application: seismic wave propagation

The spectral-finite-element method (SFEM) for modeling partial differential equations was originally developed to simulate fluid dynamics (Patera, 1984). Since then, SFEM has been widely adopted in such disparate fields as meteorology (Fournier et al., 2004), biomechanical engineering (Finol and Amon, 2001), and seismology (Tromp et al., 2008). The SFEM is so named as it demonstrates an accuracy comparable to pseudospectral methods, but is more similar to standard finite element methods (FEM) with respect to model space discretization. Thus, the SFEM examples presented here are also representative of the larger field of FEM employed frequently in a wide range of scientific/engineering computations.

Here, we discuss the implementation of a two-dimensional SFEM seismic wave propagation code on GPUs as proof of concept and as groundwork on which more advanced GPU-based SFEM codes can later be built. It should be stressed that we are primarily interested in the broader question of the GPUs performance with sparse matrix multiplication, rather than as a tool specifically for solving seismological problems. As such our goal in this section is to compare the performances of two similar GPU and CPU SFEM implementations, rather than independent CPU and GPU codes optimized for each platform. Nevertheless, such an optimized implementation has recently been brought to our attention, and we refer interested readers to Komatitsch et al. (2009).

Our SFEM implementation is based on the two-dimensional simulation described in Komatitsch et al. (2001) (Fig. 6). SFEM has proven successful as it accommodates more complex geometries and boundary conditions than pseudospectral, finite-difference, and boundary element methods, and has a reduced computational cost compared to standard FEM codes (Komatitsch and Tromp, 1999). Despite these advantages, the computational demands of SFEM simulations are often enormous. The largest seismic SFEM simulations currently employ approximately 4000 processors to simulate 14 billion grid points and require some of the world’s largest supercomputers such as the Earth Simulator in Japan (Tsuboi et al., 2005). Even relatively small simulations still contain millions of grid points (Tromp et al., 2008). Application of SFEM to body-wave scattering (Cormier, 2000; Shearer and Earle, 2004), travel-time and amplitude finite-frequency kernels (Liu and Tromp, 2006), and long-range propagation through realistic crust, (upper) mantle, and inner core heterogeneous structures (Ge et al., 2005; Niu and Chen, 2008) is hindered by the need for massive cluster computing. A parallel-GPU implementation, because of its inherent speed and lower costs, could put the power of the spectral-finite-element method into the hands of a much larger proportion of the seismological community. The method has similar advantages in ultrasonic and engineering seismic non-destructive testing (Marklein et al., 2005), where high-order accuracy and its ability to handle unusual geometries are key.

Spectral methods employ a weak form of the equations of motion for seismic waves,

\[
\int \rho \omega_j^2 s_j \frac{\partial w_j}{\partial t} \, d\mathbf{x} = - \int w_j \sigma_{ij} \frac{\partial w_i}{\partial x_j} \, d\mathbf{x} + M_i(t)w_i(\mathbf{x}'),
\]

where \( \sigma_{ij} \) are the stress tensor components, \( s_j \) are the displacement field components, \( M_i(t) \) are the source moment tensor components of the earthquake originating at location \( \mathbf{x}' \) at time, \( t \), and \( w_i \) are the components of the vector field used to generate the weak formulation. Here, Einstein summation is employed for Roman subscripts, in which repetition implies summation over the indices, while subscripted commas represent derivatives with respect to the corresponding coordinate, i.e., \( A_j = \partial A / \partial x_j \).
The modeled domain is subdivided into deformed cubic elements (see Fig. 5a for a two-dimensional representation), such that points within the unit cube, \( g \), are mapped to the points within each element, \( x \), via a set of shape functions, \( N_a(g) \).

Lagrange polynomials interpolate the state variables and their spatial derivatives within the elements,

\[
f(x(g)) = \sum_{a=0}^{n} f^a L^a(g), \quad f_i = \sum_{a=0}^{n} f^a l^a_i,
\]

where \( L^a \) is a product of two Lagrange polynomials, \( l^a = L^a(g) \), and \( f^a \) is a product of shape functions, \( N^a(g) \). The integrals in the weak formulation are approximated by

\[
\int f \, dx = \int f(x(g)) \eta \, d\eta \approx \sum_{a=0}^{n} \omega_2 \omega_1 f^a L^a, \quad (5)
\]

where \( \omega_a \) denote quadrature weights associated with the numerical integration and \( f^a \) represents the value of the Jacobian at the quadrature points, calculated from the shape functions, \( N^a(\eta) \). The quadrature points are selected using the Gauss–Lobatto–Legendre (GLL) integration rule, which yields a formulation favorable to parallel implementation (Tromp et al., 2008).

SFEM has an additional advantage for GPU implementation, namely the contribution from each element leads to a block-compressed sparse-row stiffness matrix (Fig. 5). This matrix structure can be easily manipulated to ensure coalesced read/write operations, necessary to obtain maximum GPU performance. The block-compressed sparse-row stiffness matrix is an example of the “sparse matrix linear algebra” class (Table 1) identified in the Berkeley report (Asanovic et al., 2006) (Table 3).

Our GPU implementation proceeds in the following manner:

1. Each element is assigned a thread block, with each block containing a number of threads equal to the number of nodes in each element. The elements’ individual contributions to the stiffness matrix, \( \mathbf{K} \), are calculated and stored as a block-compressed sparse-matrix format (recording the degrees of freedom associated with the rows and columns, as well as the associated values) in global memory (i.e., the GPU’s onboard memory).

   a. Each element is considered separately—no effort is made at this point to sum the contributions from nodes that appear in multiple elements. Although the entire stiffness matrix is calculated and stored in our implementation, the mesh need not be processed all at once in this manner. Instead separate regions of the mesh can be calculated individually—thereby reducing memory requirements for larger meshes. For a perfectly elastic material, \( \mathbf{K} \) is constant, in anelastic materials, however, \( \mathbf{K} \) changes as a function of the strain history (Tromp et al., 2008). Consequently, we conduct two sets of simulations: in one set \( \mathbf{K} \) is recalculated at each timestep (Tromp et al., 2008). Moreover, the higher-order accuracy of SFEM means that explicit, rather than implicit, methods are employed in the timestepping scheme (Fig. 6).

SFEM has an additional advantage for GPU implementation, namely the contribution from each element leads to a block-compressed sparse-row stiffness matrix (Fig. 5). This matrix structure can be easily manipulated to ensure coalesced read/write operations, necessary to obtain maximum GPU performance. The block-compressed sparse-row stiffness matrix is an example of the “sparse matrix linear algebra” class (Table 1) identified in the Berkeley report (Asanovic et al., 2006) (Table 3).

Our GPU implementation proceeds in the following manner:

1. Each element is assigned a thread block, with each block containing a number of threads equal to the number of nodes in each element. The elements’ individual contributions to the stiffness matrix, \( \mathbf{K} \), are calculated and stored as a block-compressed sparse-matrix format (recording the degrees of freedom associated with the rows and columns, as well as the associated values) in global memory (i.e., the GPU’s onboard memory).

   a. Each element is considered separately—no effort is made at this point to sum the contributions from nodes that appear in multiple elements. Although the entire stiffness matrix is calculated and stored in our implementation, the mesh need not be processed all at once in this manner. Instead separate regions of the mesh can be calculated individually—thereby reducing memory requirements for larger meshes. For a perfectly elastic material, \( \mathbf{K} \) is constant, in anelastic materials, however, \( \mathbf{K} \) changes as a function of the strain history (Tromp et al., 2008). Consequently, we conduct two sets of simulations: in one set \( \mathbf{K} \) is recalculated at each
The new velocity vector is calculated from than two orders of magnitude (Table 3). If instead, the matrix is recalculated rather than stored, the speedup can be more than an equivalent CPU implementation. In particular, if the stiffness matrix at each timestep reduces the overall proportion of computation times slower than running the same simulation with the stiffness matrix stored on the CPU. This is interesting as the stiffness matrix can be reconstructed on the GPU in separate sections that are processed sequentially. Such an implementation would save memory, making it feasible to run larger (albeit slightly slower) simulations than would be possible with the CPU alone.

As in the lattice-Boltzmann example, the performance gains shown assume that all nodes in the mesh are stored in GPU memory. Part of the performance increase may also be due to the fact that our SFEM implementation is two dimensional and we have confined our analysis to relatively small meshes—memory access in larger, three-dimensional meshes may incur a higher cost. Nevertheless, the three-dimensional GPU implementation of an SFEM code in Komatitsch et al. (2009) mentioned earlier was able to achieve a 25× speedup compared to an optimized CPU version. The next challenge will be to demonstrate how these simulations scale across multiple CPU–GPU systems, in order to account for more complicated three-dimensional models, with the eventual goal of scaling up to whole-earth simulations.

Table 3
SFEM performance in node updates per second versus GLL interpolation order.

<table>
<thead>
<tr>
<th>GLL order</th>
<th>CPU (nodes/second)</th>
<th>GPU (nodes/second)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 × 4</td>
<td>5120</td>
<td>570,000</td>
<td>11×</td>
</tr>
<tr>
<td>5 × 5</td>
<td>3790</td>
<td>353,000</td>
<td>93×</td>
</tr>
<tr>
<td>6 × 6</td>
<td>2850</td>
<td>213,000</td>
<td>74×</td>
</tr>
</tbody>
</table>

Simulation speeds are based on 10,000 timesteps performed on an 11 × 11 element grid. The stiffness matrix, K, is either recalculated at each timestep (K recalculated), or calculated once and stored in memory (K stored). GPU results are from a single 2.66 GHz Intel Core 2 Quad processor. GPU times based on an NVIDIA GeForce 8800 GTX with a clock rate of 1.188 GHz.

The GPU again delivers a significant performance increase over the equivalent CPU implementation. In particular, if the stiffness matrix is recalculated rather than stored, the speedup can be more than two orders of magnitude (Table 3). If instead, the matrix is stored in memory the speedup is less (7–19 times), but still not unsubstantial. We attribute the differences in speedups between the two versions to the fact that recalculating the stiffness matrix at each timestep reduces the overall proportion of computation time spent on memory access. We also note that recalculating the stiffness matrix at each timestep on the GPU is only a factor of 2–3 times slower than running the same simulation with the stiffness matrix stored on the CPU. This is interesting as the stiffness matrix can be reconstructed on the GPU in separate sections that are processed sequentially. Such an implementation would save memory, making it feasible to run larger (albeit slightly slower) simulations than would be possible with the CPU alone.

As in the lattice-Boltzmann example, the performance gains shown assume that all nodes in the mesh are stored in GPU memory. Part of the performance increase may also be due to the fact that our SFEM implementation is two dimensional and we have confined our analysis to relatively small meshes—memory access in larger, three-dimensional meshes may incur a higher cost. Nevertheless, the three-dimensional GPU implementation of an SFEM code in Komatitsch et al. (2009) mentioned earlier was able to achieve a 25× speedup compared to an optimized CPU version. The next challenge will be to demonstrate how these simulations scale across multiple CPU–GPU systems, in order to account for more complicated three-dimensional models, with the eventual goal of scaling up to whole-earth simulations.

5. Least-squares minimization—example application: magnetic scanning force microscopy

Least-squares minimization (LSM) is used throughout the geosciences, and in many other fields, as a tool for data fitting, signal processing, and inverse theoretical methods. Here least-squares minimization techniques are used to interpret magnetic field intensity data, obtained from scanning force microscopy and related methods (e.g., SQUID and Hall probe microscopy Kirtley and Wikswo, 1999; Chang et al., 1992). A scanning force microscope contains a micron-scale cantilever with a sharp tip, similar to a phonograph arm and stylus. The cantilever tip is scanned across the sample surface and its vertical displacement inferred from the phase shift in a laser reflected off the cantilever. Magnetic force microscopy (MFM) is conducted by coating the tip with a magnetic material. The cantilever responds to the net magnetic force experienced between its tip and the sample (Frandsen et al., 2004). The most probable distribution of magnetic dipoles is obtained from the measured magnetic forces via a least-squares fit (Weiss et al., 2007). Due to the long range forces that act between each dipole and the cantilever, the least-squares minimization problem gives rise to a dense matrix in which every element interacts with every other element, and is thus representative of the “dense linear algebra” class (Table 1) identified in the Berkeley report (Asanovic et al., 2006).

Traditional paleomagnetic studies use bulk rock samples that often contain different magnetic minerals in a multitude of grain sizes. These various mineral phases and grains often form...
throughout the rock’s history, recording conflicting magnetizations, which confounds interpretation of the overall remanence. To address this problem, researchers are obliged to conduct a litany of time consuming rock magnetic experiments designed to identify various mineral phases and their abundances within a given sample. MFM offers a means of overcoming many of these issues by providing grain-scale information of the rock sample. However, to date, many of the benefits of this approach remain unrealized due to the amount of time needed for data processing.

GPU implementation of this problem has the potential to reduce analysis times from several days to hours or less, without requiring external supercomputing facilities. This would greatly enhance the capabilities of magnetic scanning force microscopy, allowing rapid processing of larger, higher resolution scans and enhancing the capabilities of magnetic scanning force microscopy, reducing analysis times from several days to hours or less, without issues by providing grain-scale information of the rock sample. MFM offers a means of overcoming many of these issues.

For linear problems (e.g., if dipole locations are fixed and only point sources of variable intensities and directions, \(x\), i.e., \(M_i(x) = \sum m_i \delta(x - x^i)\), where \(\delta(x)\) is the Dirac delta function. Due to the noisy nature of the magnetic intensity data, the magnetic field predicted by the dipole distribution will only approximately match the measured data, \(B_i\), with a misfit residual, \(R_i\), of

\[
R_i(x^d) = \sum_B |G(x, x^d) M_j(x)| - B_i(x^d).
\]

Thus, the goal is to minimize the square of \(R_i(x)\) with respect to predetermined degrees of freedom, \(\xi\), i.e.,

\[
\frac{\partial}{\partial \xi} \sum_{x^d} R_i(x^d) R_i(x^d) = 0.
\]

For linear problems (e.g., if dipole locations are fixed and only their magnitudes allowed to vary or if an overall dipole direction is sought), Eq. (9) is solved by finding the single value decomposition (SVD) of a matrix equation, \(A_j \delta_j = B_i\), where \(A_j\) expresses the relationship between the degrees of freedom and the magnetic force and \(B_i\) are the observed magnetic intensities from the MFM scan. If dipole positions are also allowed to vary, the problem becomes non-linear. In this case, the best fit is found by constructing a Hessian matrix for the problem, and then finding the minima using a non-linear method (e.g., Newton’s method). The analysis given here is confined to the linear case, although we plan to apply GPGPU computation to the nonlinear inverse problem in future work.

We implement the SVD using a basic one-sided Jacobi algorithm (Drmac and Veselić, 2008a, b). Although faster SVD methods are available, the one-sided Jacobi algorithm is chosen due to its high accuracy, ease of parallel implementation, and relative simplicity. The Jacobi method calculates the SVD of a matrix, \(G\), by constructing an orthogonal matrix, \(V\), that diagonalizes \(A = G^T G\). If \(V^T AV = W^2\), then the singular value decomposition is \(G = U W V^T\), where \(W\) is a diagonal matrix and \(U\) is orthogonal. In practice, the SVD is constructed implicitly, through the successive application of Jacobi rotations, \(\Omega\),

\[
G^0 = G, \\
G^0 = G^2 \Omega,
\]

in which the rotations are chosen to orthogonalize column pairs from the matrix \(G^0\). After several iterations through all possible column-pair combinations, the algorithm is halted once \((G^n)^T G^n\) is diagonal to numerical precision, at which point \(G^n = UW\). The SVD is then obtained from \(W = \sqrt{(G^n)^T G^n}\), \(V^T = (G^n)^{-1} G^n\), and \(U = G^n W^{-1}\).

For a matrix \(G\), with \(m\) rows and \(n\) columns, our GPU implementation of the one-sided Jacobi method proceeds in the following manner:

1. The matrix \(G\) is copied into the GPUs global memory.
2. For each “sweep” through all possible column pair combinations:
   (a) The algorithm matches column pairs using the round-robin ordering described in Zhou and Brent (1997) (Fig. 7). For each distinct round-robin column-pair ordering (\(n - 1\) in total):
      (i) Each of the column pairs (\(n/2\) in total) is assigned to a GPU thread block. Each block is given a fixed number of threads (128 was found to be optimal for the matrix sizes examined). (ii) The entries of \(A = (G^n)^T G^n\) corresponding to the products of the two columns \((A_{ij}, A_{ji}, \text{ and } A_{ij})\) are calculated, and the rotation angle, \(\theta\), for \(G^n_i\) and \(G^n_j\) is

![Fig. 7](image-url) We assign \(n/2\) thread blocks to pairs of matrix columns, and perform Jacobi pivoting on each in parallel. Subsequent combinations of column pairs are found by rotating column indexes as shown by arrows (round robin ordering of Zhou and Brent, 1997). This process is repeated \(n - 1\) times per “Jacobi sweep”, thereby rotating all \((n - 1)/2\) possible combinations of column pairs.
found. The column dot products are calculated in parallel, with each thread summing 1/128th of the column entries. The results are then summed across all threads, the value of $\theta$ calculated by a single thread, and then distributed to the others in its block.

(iii) The column pairs are orthogonalized:

$$G_{i}^{j+1} = \cos(\theta)G_{i}^{j} + \sin(\theta)G_{j}^{j},$$

$$G_{j}^{j+1} = \cos(\theta)G_{j}^{j} - \sin(\theta)G_{i}^{j}.$$

with each thread again operating on 1/128th of the rows in the two columns.

(b) The magnitudes of the off-diagonal components of $(G^T)^{-1}G^T$ are estimated from the column pair products at the start of each sweep. If the error estimate indicates the matrix is diagonal to machine precision, the Frobenious norm of the off-diagonal components of the matrix $A$ is calculated directly at the end of the sweep. The algorithm is halted once the Frobenious norm is below the desired tolerance.

The performance of our GPU SVD implementation for random square matrices is compared to the performance of two other SVD routines running on the same machine in Table 4. The first is a single processor implementation based on JAMA/C+++, a C++ implementation of the Java Matrix Library by Mathworks and NIST. The second uses the Matlab 7 command "svd", which employs LAPACK and BLAS routines. The GPU implementation delivers performance gains of up to 24× over the Matlab command and more than 100× over the JAMA/C++ implementation. Moreover, it is worthwhile noting that both of these routines employ algorithms based on QR decomposition, which are generally faster, though less accurate than the basic Jacobi SVD employed on the GPU. As with the other routines discussed in this paper, these results are contingent on the matrix being stored in the GPU memory (the maximum matrix size for the nVIDIA GeForce 8800 GTX graphics card is approximately 200,000,000 elements). Nevertheless, larger matrices can be processed using block-Jacobi methods (e.g. Hari, 2005), and we anticipate that similar performance gains will be sustained in larger matrices, as data transfer between the CPU and GPU occupies a negligible percentage of the overall calculation time (~0.16% in the 1024 × 1024 matrix SVD and ~0.04% in the 2048 × 2048 matrix SVD). It should also be noted that our current implementation is relatively unsophisticated, and has yet to incorporate many recent developments in accelerating the Jacobi SVD (e.g., Mascarenhas, 1995; Hari, 2005; Drmač and Veselić, 2008b). Admittedly, it is not yet evident how well these techniques developed for CPU implementations will translate to the GPU setting. Nevertheless, there is ample reason to believe that additional improvement is possible beyond what is given here.

Fig. 8a shows part of an MFM scan, reproduced with permission from Fig. 1b of Zhu et al. (2003). We find the dipole distribution (Fig. 8b, c) by calculating the singular value decomposition for a 120 by 120 grid of observed $B_z$ values, seeking average dipole directions at one in every four points inside the grid. The entire image is constructed by repeatedly finding the product of the SVD with overlapping regions from the original scan data, and recording the average result for each location. In total the entire process takes approximately an hour and a half. The speed of the singular value decomposition makes it feasible to conduct further operations to refine the dipole locations—for example, by thresholding the final image to reveal the most likely dipole locations and repeating the singular value decomposition. Once again the speeds reported for this simulation is entirely due to the GPU, and it is highly likely that more sophisticated techniques will deliver even greater performance gains. Obvious methods for improvement involve the use of suitable preconditioners and/or sparse matrix techniques (for example as employed in Weiss et al., 2007).

6. Conclusion

Many simulations in science and engineering, and in particular in the Earth sciences, are described in terms of simple local interactions that give rise to complex emergent macroscopic behavior. Graphics processing units (GPUs) as found on standard graphics cards, designed to implement a single set of instructions over multiple data sources simultaneously, are well suited to these types of problems. While previously GPU computation has been largely confined to specialized applications that could be cast in terms of graphics manipulations, the recent release of compilers for general purpose GPU computing has made the power of these processors more widely available.
Here, we have demonstrated that GPU implementations deliver substantial performance increases to three geophysical simulations: a lattice–Boltzmann simulation of geofluid dynamics; a spectral-finite-element method for seismic wave propagation simulations; and a least squares solver for interpreting magnetic-force-microscopy data. The example applications each involve different computing classes (structured grid, sparse linear algebra, and dense linear algebra) deemed fundamental to scientific and engineering programming, and hence may be considered indicative of the performance of GPU computing to a much broader set of problems in many research fields. The GPU implementations were found to deliver performance gains of 10×–30× compared to equivalent CPU implementations.

There are some clear limitations to GPU computing, in particular (a) the amount of memory on most graphics cards is (currently) somewhat limited, although cards with larger memories specifically designed for GPGPU computing are available, (b) the speed of data transfer between the GPU and host CPU is (currently) low, and (c) the degree of complexity in the individual elements being simulated is restricted. These limitations may in part be overcome as these relatively nascent systems mature from their graphics processing origins. Other inefficiencies in general purpose GPU systems will also undoubtedly be overcome by the development of new algorithms specifically optimized for these types of computing environments. Finally, the simplest way to resolve potential limitations of individual GPU performance compared to existing parallel CPU systems, may be through the development of GPU clusters. Indeed, our future work will look at sustaining the performance gains achieved in this paper over several parallel-linked GPUs, opening the way for efficient programs designed to take advantage of GPU clusters.

Although general purpose GPU computing should not be regarded as a general panacea, it nevertheless offers several exciting advantages over traditional CPU computing for certain classes of problems. Particularly for many of the simulations encountered in geology and the wider scientific and engineering community, GPUs may present supercomputing opportunities that would otherwise be out of reach to all but the most advanced computing facilities. Such facilities, would in turn, however, also benefit from cost-effective multi-GPU cluster implementations with thousands of GPUs, each containing hundreds of processors. Such computing systems may then allow new types of simulations at unprecedented scales, resolutions, or computing speeds.

Acknowledgments

We would like to thank Joshua Feinberg, Justin Reavenaugh, Erich Frahm and Casey Battaglino for their assistance and helpful suggestions while writing this paper. We also gratefully acknowledge the use of resources from the University of Minnesota Supercomputing Institute (MSI), and thank nVIDIA for supplying the 8800 GTX graphics card. Martin Saar thanks the George and Orpha Gibson endowment for its generous support of the Hydrogeology and Geofluids research group. In addition, this material is based upon support by the National Science Foundation (NSF) under Grant nos. DMS-0724560 and EAR-0510723. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. We thank the anonymous reviewers for their helpful comments that improved this paper.

References


