PARALLEL COMPUTING EXPERIENCES WITH CUDA

Michael Garland
Scott Le Grand
John Nickolls
NVIDIA

Joshua Anderson
Iowa State University and Ames Laboratory

Jim Hardwick
TechniScan Medical Systems

Scott Morton
Hess

Everett Phillips
Yao Zhang
University of California, Davis

Vasily Volkov
University of California, Berkeley

THE CUDA PROGRAMMING MODEL PROVIDES A STRAIGHTFORWARD MEANS OF DESCRIBING INHERENTLY PARALLEL COMPUTATIONS, AND NVIDIA'S TESLA GPU ARCHITECTURE DELIVERS HIGH COMPUTATIONAL THROUGHPUT ON MASSIVELY PARALLEL PROBLEMS. THIS ARTICLE SURVEYS EXPERIENCES GAINED IN APPLYING CUDA TO A DIVERSE SET OF PROBLEMS AND THE PARALLEL SPEEDUPS OVER SEQUENTIAL CODES RUNNING ON TRADITIONAL CPU ARCHITECTURES ATTAINED BY EXECUTING KEY COMPUTATIONS ON THE GPU.

With the transition from single-core to multicore processors essentially complete, virtually all commodity CPUs are now parallel processors. Increasing parallelism, rather than increasing clock rate, has become the primary engine of processor performance growth, and this trend is likely to continue. This raises many important questions about how to productively develop efficient parallel programs that will scale well across increasingly parallel processors.

Modern graphics processing units (GPUs) have been at the leading edge of increasing chip-level parallelism for some time. Current NVIDIA GPUs are many-core processor chips, scaling from 8 to 240 cores. This degree of hardware parallelism reflects the fact that GPU architectures evolved to fit the needs of real-time computer graphics, a problem domain with tremendous inherent parallelism. With the advent of the GeForce 8800—the first GPU based on NVIDIA's Tesla unified architecture—it has become possible to program GPU processors directly, as massively parallel processors rather than simply as graphics API accelerators.

NVIDIA developed the CUDA programming model and software environment to let programmers write scalable parallel programs using a straightforward extension of the C language. The CUDA programming model guides the programmer to expose substantial fine-grained parallelism sufficient for utilizing massively multithreaded GPUs, while at the same time providing scalability across the broad spectrum of physical parallelism available in the range of GPU devices. Because it provides a fairly simple, minimalist abstraction of parallelism and inherits all the well-known semantics of C, it lets programmers develop massively parallel programs with relative ease.

In the year since its release, many developers have used CUDA to parallelize and accelerate computations across various problem domains. In this article, we survey some experiences gained in applying CUDA to a diverse set of problems and the parallel speedups attained by executing key computations on the GPU.

0272-1732/08/$20.00 © 2008 IEEE Published by the IEEE Computer Society

```
void saxpy(uint n, float a,
           float *x, float *y)
{
   for (uint i = 0; i<n; ++i)
      y[i] = a*x[i] + y[i];
}


void serial_sample ()
{
   // Call serial SAXPY function
   saxpy (n, 2.0, x,y);
}
```

(a)

```
__global__ void saxpy(uint n, float a,
                      float *x, float *y)
{
   uint i = blockIdx.x*blockDim.x
            + threadIdx.x;

   if( i<n ) y[i] = a*x[i] + y[i];
}

void parallel_sample()
{
   // Launch parallel SAXPY kernel
   // using ⌈n/256⌉ blocks of 256
   // threads each
   saxpy<<<ceil(n/256),256>>>(n, 2, x, y);
}
```

(b)

Figure 1. Parallel programming with CUDA: serial (a) and parallel (b) kernels for computing $\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}$.

CUDA parallel programming model

The CUDA parallel programming model emphasizes two key design goals. First, it aims to extend a standard sequential programming language, specifically C/C++, with a minimalist set of abstractions for expressing parallelism. Broadly speaking, this lets the programmer focus on the important issues of parallelism—how to craft efficient parallel algorithms—rather than grappling with the mechanics of an unfamiliar and complicated language. Second, it is designed for writing highly scalable parallel code that can run across tens of thousands of concurrent threads and hundreds of processor cores. This is essential because the physical parallelism of current NVIDIA GPUs ranges from eight processor cores and 768 thread contexts up to 240 processor cores and 30,720 thread contexts. The CUDA model naturally guides the programmer to write parallel programs that transparently and efficiently scale across these different levels of parallelism.

A CUDA program is organized into a host program, consisting of one or more sequential threads running on the host CPU, and one or more parallel kernels that are suitable for execution on a parallel processing device like the GPU. A kernel executes a scalar sequential program on a set of parallel threads. The programmer organizes these threads into a grid of thread blocks. The threads of a single thread block are allowed to synchronize with each other via barriers and have access to a high-speed, per-block shared on-chip memory for interthread communication. Threads from different blocks in the same grid can coordinate only via operations in a shared global memory space visible to all threads. CUDA requires that thread blocks be independent, meaning that a kernel must execute correctly no matter the order in which blocks are run, even if all blocks are executed sequentially in arbitrary order without preemption. This restriction on the dependencies between blocks of a kernel provides scalability. It also implies that the need for global communication or synchronization amongst threads is the main consideration in decomposing parallel work into separate kernels.

The details of the CUDA programming model are available in NVIDIA's CUDA Programming Guide (www.nvidia.com/CUDA) and related literature.[1] Figure 1 shows some basic features of parallel programming with CUDA. It contains straightforward implementations, both sequential and parallel, of the SAXPY routine defined by the BLAS linear algebra library. Given vectors $\mathbf{x}$ and $\mathbf{y}$ containing n floating-point numbers, it performs the update $\mathbf{y} \leftarrow \alpha \mathbf{x} + \mathbf{y}$. The serial implementation is a simple loop that computes one element of $\mathbf{y}$ in each iteration. The parallel kernel

effectively executes each of these independent iterations in parallel, assigning a separate thread to compute each element of **y**. The **__global__** modifier indicates that the procedure is a kernel entry point, and the extended function call syntax **saxpy <<<B, T>>>(…)** is used to launch the kernel **saxpy()** in parallel across *B* blocks of *T* threads each. Each thread of the kernel determines which element it should process from its integer thread block index (**blockIdx.x**), its index within its block (**threadIdx.x**), and the total number of threads per block (**blockDim.x**).

This example demonstrates a common parallelization pattern, where a serial loop with independent iterations can be executed in parallel across many threads. In the CUDA paradigm, the programmer writes a scalar program—the parallel **saxpy()** kernel—that specifies the behavior of a single thread of the kernel. This lets CUDA leverage the underlying C language, with only a few small additions, such as the built-in thread and block index variables.

The SAXPY kernel is also a simple example of data parallelism, where parallel work is decomposed to match result data elements. Although different thread blocks or different threads of a kernel can potentially execute entirely different code, such task parallelism does not generally scale as well as data parallelism. Moreover, data-parallel kernels typically expose substantially more fine-grained parallelism than task-parallel kernels and, therefore, generally can take best advantage of the GPU architecture.

NVIDIA Tesla GPU architecture

NVIDIA designed its Tesla unified graphics and computing architecture[2] to accelerate parallel programs written in the CUDA programming model. It is built around a fully programmable processor array, organized into a number of SM multithreaded multiprocessors, each of which contains eight scalar SP processor cores. In contrast to earlier generations of GPUs, threads executing on the SP cores have access to the full range of instructions programmers expect in any general-purpose processor core. In particular, memory is accessed through load/store instructions supporting arbitrary address arithmetic, and it fully supports both floating-point and integer data types.

Figure 2 shows the Tesla architecture of a GeForce GTX 280 or Tesla T10 GPU with 240 SP streaming processor cores, organized in 30 SM streaming multiprocessors. Each multithreaded SP core executes up to 128 concurrent coresident threads sharing a register file of 2,048 entries; in total, the GPU executes up to 30,720 concurrent threads. The earlier GeForce 8800 GTX GPU provides 16 SMs with 1,024 registers per SP core and supports a maximum of 12,288 concurrent threads. Thread creation, scheduling, and resource management are performed entirely in hardware. The cost of creating and destroying threads is negligible, and there is effectively no overhead involved in thread scheduling. Each thread of a CUDA program is mapped to a physical thread resident in the GPU, and each running thread block is physically resident on a single SM. The SM multiprocessor supports efficient communication among threads in a block by providing an extremely lightweight barrier synchronization facility—CUDA's barrier intrinsic translates into a single instruction—and 16 Kbytes per SM of on-chip shared memory that has low access latency and high bandwidth, similar to an L1 cache.

The Tesla architecture is designed to operate on massively parallel problems. The deep multithreading provides substantial latency tolerance, allowing resources to be dedicated toward throughput rather than large caches. As we discussed earlier, experience shows that data-parallel software design is frequently the best approach for managing this level of fine-grained parallelism. The threads of a data-parallel kernel will often be following substantially similar paths of execution. Consequently, the Tesla architecture is optimized for this case. The Tesla SM employs a single instruction, multiple thread (SIMT) architecture,[1,2] where a block's threads are grouped into warps containing 32 threads each. A warp's threads are free to follow arbitrary and independent execution paths, involving any number of branches, but can collectively execute only a single instruction at any

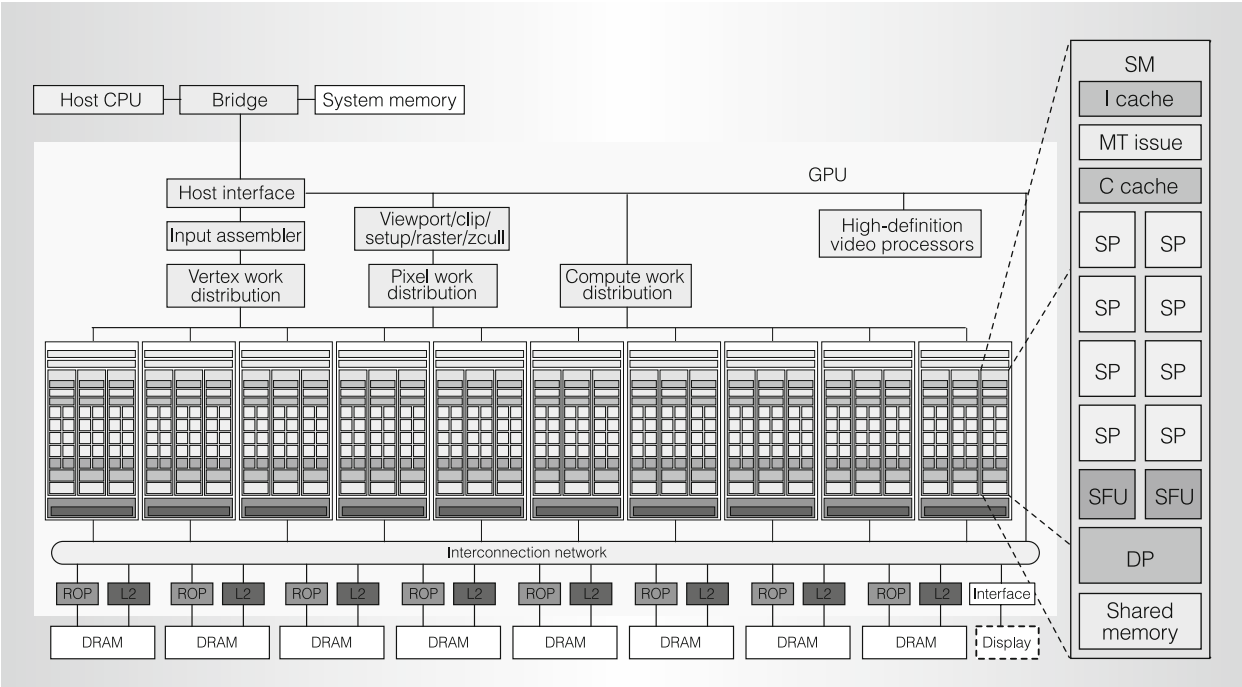

Figure 2. Tesla unified graphics and computing architecture of a GeForce GTX 280 or Tesla T10 GPU with 240 SP streaming processor cores, organized in 30 SM multithreaded multiprocessors. Each multithreaded SP core executes up to 128 concurrent threads; the GPU executes up to 30,720 concurrent threads.

particular instant. Thus, threads in a warp that execute different code paths (that is, that diverge) must wait in turn while other threads of the warp execute the instructions they wish. Divergence and reconvergence are managed in hardware. On the other hand, in the common case where all threads of a warp are executing the same instruction, all the warp's threads can execute that instruction at the same time. This lets the hardware achieve substantial efficiencies when executing typical data-parallel programs, while at the same time making the SIMT execution of threads transparent to the programmer and providing the flexibility of a scalar computational model.

Application experience with CUDA

Many applications consist of a mixture of fundamentally serial control logic and inherently parallel computation. Furthermore, these parallel computations are frequently data-parallel in nature. This directly matches the program model that CUDA adopts, namely a sequential control thread capable of launching a series of parallel kernels. The use of parallel kernels launched from a sequential program also makes it relatively easy to parallelize an application's individual components, rather than requiring a wholesale rewriting of the entire application.

Many applications—both academic research and industrial products—have been accelerated using CUDA to achieve significant parallel speedups. Such applications fall into a variety of problem domains, including machine learning,[3] database processing,[4] bioinformatics,[5,6] financial modeling, numerical linear algebra, medical imaging,[7] and physical simulation, among others. Of the many available examples, we survey a few representative cases.

Molecular dynamics

Molecular dynamics is a simulation technique widely used in physics, chemistry, biology, and related fields. Its goal is to compute the movement of a number of atoms, beginning in some initial configura-

tion, and track their trajectory over specified time intervals. Molecular dynamics is a versatile tool that scientists can use to calculate materials' static and dynamic properties, study methods of crystal growth, or examine mechanisms by which proteins perform their functions, to name only a few of its uses.[8] Figure 3 shows an example snapshot from a bead-spring polymer simulation with each of 64,017 particles displayed as a sphere.[9]

Molecular dynamics simulations are inherently parallel computations and are ideally suited to the CUDA programming model. During each time step of the simulation, the program must calculate the forces acting on all atoms and use the resulting forces to integrate atom positions and velocities forward to the next step. The different tasks required in a time step can each be implemented in a separate kernel. Because each atom can be processed independently of the others during a single time step, it is natural to map each atom to a single thread. Thus, the application naturally provides the large amount of fine-grained parallelism for which the GPU is designed, and several molecular dynamics[10] and molecular modeling[11,12] codes have been successfully accelerated with CUDA.

A typical molecular dynamics simulation of the polymer model in Figure 3 would be run for 18 million time steps, and during each time step, every atom can be processed independently of the others. Performing all this work sequentially can be expensive; a serial calculation using a single core of a 2.4-GHz Opteron 280 CPU can calculate about 6.46 time steps per second (tps) and would take about 32 days to complete the entire run. Usually, scientists perform jobs of this size using parallel software such as LAMMPS,[13] possibly completing it in a single day if executed on 32 processor cores in a cluster of Opteron 280 nodes connected by InfiniBand. Implemented in CUDA and executing on a single GeForce 8800 GTX, Highly Optimized Object-Oriented Molecular Dynamics (HOOMD; www.ameslab.gov/hoomd) can execute the same simulation with a performance of 203 tps, equivalent to that of LAMMPS using 32 CPU cores.[10] A multi-GPU implementation is currently under development, and the performance is expected to scale well with the number of GPUs in the workstation.

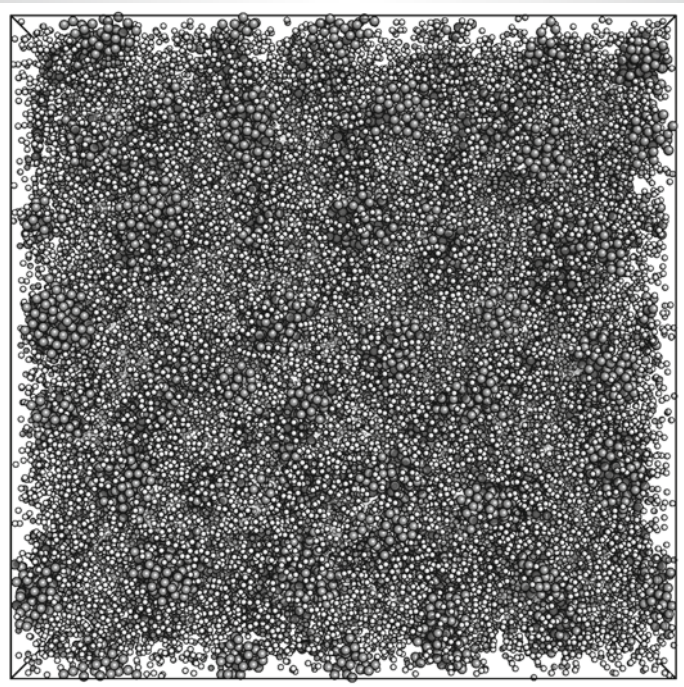

Figure 3. Snapshot of a bead-spring polymer simulation with 64,017 particles.

Due to the excellent scalability of Tesla-architecture GPUs, HOOMD's performance scales linearly with particle count, from systems as small as 5,000 particles to those consuming all available memory. Figure 4 shows the time taken to execute just the pair force summation kernel for the polymer system with varying numbers of particles (N) and the speedup relative to an optimized serial CPU implementation running on a single-core 3.0-GHz Xeon 80546K processor.[10] The GPU is very efficient at the pair force calculation kernel, offering speedup factors of approximately 60 times for large systems. Furthermore, it is able to accelerate the entire application-level simulation for the problem shown in Figure 3 by a total of 32 times.

In HOOMD, the different tasks required in a time step are each implemented in a separate kernel. The host program performs

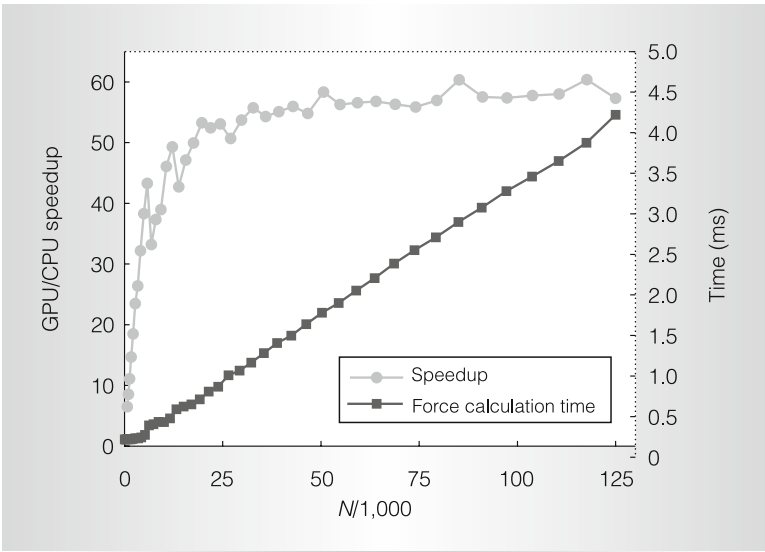

Figure 4. Scaling and parallel speedup of GPU implementation over single-CPU core for the pair force calculation kernel, across many problem sizes.

all memory management and other setup tasks and then calls the GPU kernels to perform the simulation. Each kernel breaks its task down further in a data-parallel fashion, with each thread mapped to a single particle. This is a natural choice for molecular dynamics, where all particles are updated independently from each other during a single time step, and is a natural fit for CUDA. When executing the polymer model simulations, completing each time step in HOOMD requires calling a number of kernels in sequence. First, the neighbor list data structure must be updated, although this doesn't strictly need to be done at every step. Then it can be used to compute pair forces. After that, bond forces are computed, and the system is integrated forward to the next time step. Each of these tasks requires calling one or more kernels totaling five to eight for each time step, depending on whether the neighbor lists have been updated.

The GeForce 8800 GTX offers a peak multiply-add rate of 345 Gflops and a memory throughput of 86 Gbytes/s. Making the most of these resources is key in creating high-performance GPU kernels. For most calculations in molecular dynamics, this means optimizing the memory accesses first because there are relatively few floating-point operations for each memory read. With CUDA, it is straightforward to write a kernel with a simple memory access pattern that achieves near-peak performance. Such kernels for performing the integration step are nearly as simple as the previous SAXPY example and achieve near 70 Gbytes/s. The pair force computation involves more complicated and random memory access patterns. Utilizing the 1D texture cache together with a space-filling curve-based data-reordering technique, the pair force computation can execute at close to peak bandwidth, utilizing 76 Gbytes/s and 130 Gflops.[10] Achieving these performance levels requires no assembly language programming or complicated code transformations, just writing simple and readable data-parallel C code.

Folding@Home is a popular application for molecular dynamics that runs on many hardware platforms.[14] The performance bottleneck for this application involves three N-body calculations over all $O(N^2)$ unique pairs of atoms. The first two of these calculations are independent and can be merged into one calculation, but the third calculation depends on results generated by the second. This leaves us with two such loops that consume approximately 75 percent of this application's computation. Previous GPU-based implementations have calculated all pair-wise interactions due to the complexity of tracking only unique pairs in a streaming parallel implementation. In contrast, the CUDA implementation operates only on unique pairs. It blocks the calculation on N bodies into a series of $p \times p$ swaths, where p is the warp width. This lets the p threads within each warp each read one atom's worth of data into their register space and then operate on p atoms that were read into on-chip shared memory. Furthermore, because all the threads in a warp are guaranteed to execute synchronously together, each thread can interact with one atom's data in shared memory at a time over p iterations without any need for additional synchronization.

Folding@Home presents an excellent opportunity to compare the performance of several different architectures on the same application. Figure 5 plots the performance

of the CUDA implementation of the Folding@Home energy kernel on a GeForce 8800M GTS (a laptop GPU with 64 SPs), a GeForce 8800 GTS (a desktop GPU with 128 SPs), and the latest GeForce GTX 280 (a high-end GPU with 240 SPs). The CUDA programming model was specifically designed to enable the design of scalable parallel computations, and here we see clear evidence of strong scaling across a range of processors with substantially different levels of physical parallelism. To place their performance in context, the graph also shows the performance of this simulation on a Core2 Duo CPU, the Cell processor of a Sony PlayStation 3, and an ATI Radon 3870 GPU. The algorithmic flexibility that CUDA provides, and the architectural features that it exploits—specifically shared memory—allow the CUDA implementation on the GTX 280 to run 3.6 times faster than the Brook-based ATI implementation and 6.7 times faster than the Cell implementation.

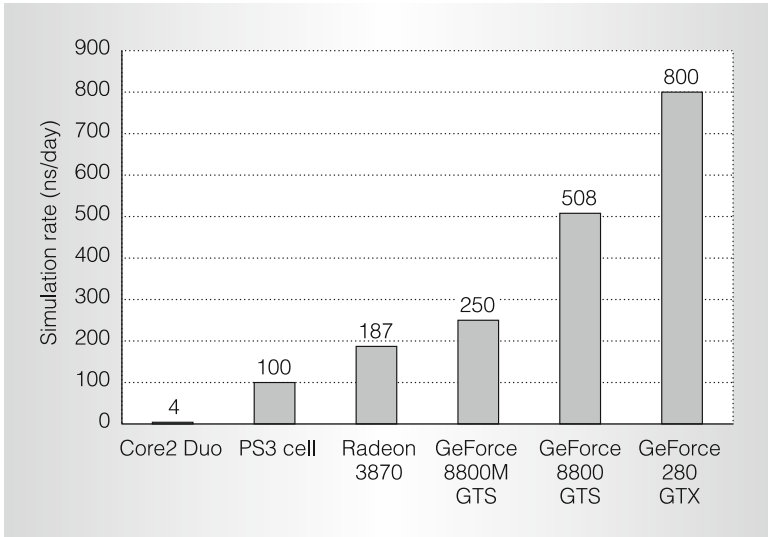

Figure 5. Performance of Folding@Home energy kernel on various platforms.

Numerical linear algebra

Dense matrix-matrix multiplication, particularly as provided by the BLAS library GEMM routines, is one of the fundamental building blocks of numerical linear algebra algorithms. It is also a natural fit for CUDA and the GPU because it is inherently parallel and can naturally be expressed as a blocked computation.

Volkov and Demmel[15] describe the design of a custom SGEMM matrix-matrix multiplication kernel. Figure 6 summarizes the performance of their algorithm running on a GeForce 8800 GTX and Intel's Math Kernel Library (MKL) 10.0, running on a 2.4-GHz Core2 Quad Q6600 ("Kentsfield") processor. This algorithm, which operates on single-precision floating-point numbers, achieves up to 206 Gflops—a 2.9 times improvement over the highest rate achieved by the Core2 Quad—and roughly 60 percent of the GeForce 8800 GTX peak multiply-add rate.

Writing a basic dense matrix-matrix multiplication kernel is a fairly simple exercise (see the CUDA Programming Guide for detail). Achieving this high level of performance, on the other hand, requires more careful optimization. Volkov and Demmel use a block algorithm similar to those used for vector computers, using GPU registers and per-block shared memory to store the data blocks. As the GPU has an unusually large register file, registers can be used as the primary scratch space for the computation. Furthermore, assigning small blocks of elements to each thread, rather than a single element to each thread, boosts efficiency much as strip-mining boosts efficiency on vector machines. Finally, the nonblocking nature of loads on the GPU

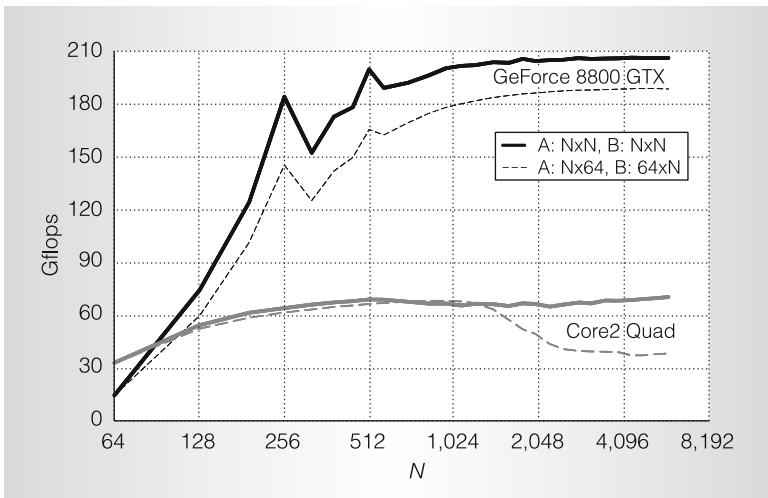

Figure 6. Performance on dense matrix-matrix multiplication (SGEMM).

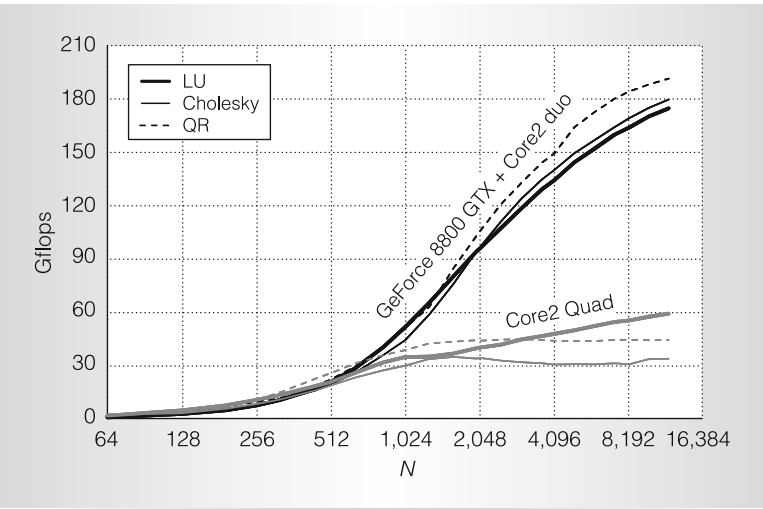

Figure 7. Performance on dense LU, Cholesky, and QR factorization for **N** × **N** matrices.

makes it possible to do software prefetching, which is useful for hiding memory latency. Ryoo and colleagues explored the benefits of blocking and loop unrolling, producing a matrix multiplication kernel executing at 91 Gflops.[16] The gap with the performance achieved by Volkov and Demmel clearly demonstrates the utility of the additional optimizations they employed, namely storing as much data as possible in registers, using larger data blocks, and strip-mining multiple elements onto each thread.

Matrix factorizations are widely used to solve systems of linear equations and linear least-square problems, and among the many available factorization schemes the LU, Cholesky, and QR factorizations are most common. Like matrix-matrix multiplication, these factorizations can be implemented using blocked algorithms that do most of their work with bulk matrix-matrix multiplies that have high arithmetic intensity and expose substantial amounts of data and thread-level parallelism. The remaining work lies in factorizing tall and skinny matrices (called *panels*) and pivoting. Panel factorizations involve many small fine-grained operations that might not offer sufficient parallelism to run efficiently on the GPU. Therefore, it is often more efficient to perform panel factorization on the CPU while using the GPU to perform the inherently parallel parts of the computation, particularly since these computations can then be overlapped. For efficient partial pivoting in LU factorization, it is necessary to arrange accesses to memory to have a minimal stride, which can be done by keeping matrices transposed in the GPU memory.

Figure 7 compares the performance of these factorization routines running on a GeForce 8800 GTX in a 2.67-GHz Core2 Duo system, versus performing the factorization entirely on a 2.4-GHz Core2 Quad with MKL. The CUDA factorization code achieves rates of up to 190 Gflops, which approaches the peak rate sustained by matrix-matrix multiplication itself. For matrix dimensions of around 300, the problem becomes large enough to leverage the GPU parallelism and overcome the overhead of CPU-GPU coordination and data movement. For larger matrices, the CUDA factorization running on the GPU and Core2 Duo is up to 5.5 times faster than the MKL factorization code running on the Core2 Quad.

Medical imaging

TechniScan Medical Systems has been developing advanced inverse-scattering algorithms to generate 3D volumetric images of the breast with ultrasound for several years. Unlike conventional ultrasound imaging, in which reflected ultrasound is used to form images, inverse-scattering uses ultrasound transmitted through, refracted by, and scattered by breast tissue to generate high-resolution speed and attenuation of sound images.[17] Pending FDA clearance, these images are intended to provide additional information to improve care and outcome in breast disease treatment (see Figure 8).

In the last five years, CPUs have become fast enough to run these algorithms in several hours on a small high-performance computing cluster inside the TechniScan Whole Breast Ultrasound (WBU) scanner. Radiologists and women's health providers, however, would like images in minutes so that they can conduct the exam and discuss results with their patients in the same visit. TechniScan has investigated porting the inverse-scattering algorithm to FPGAs, digital signal processors (DSPs), and IBM's

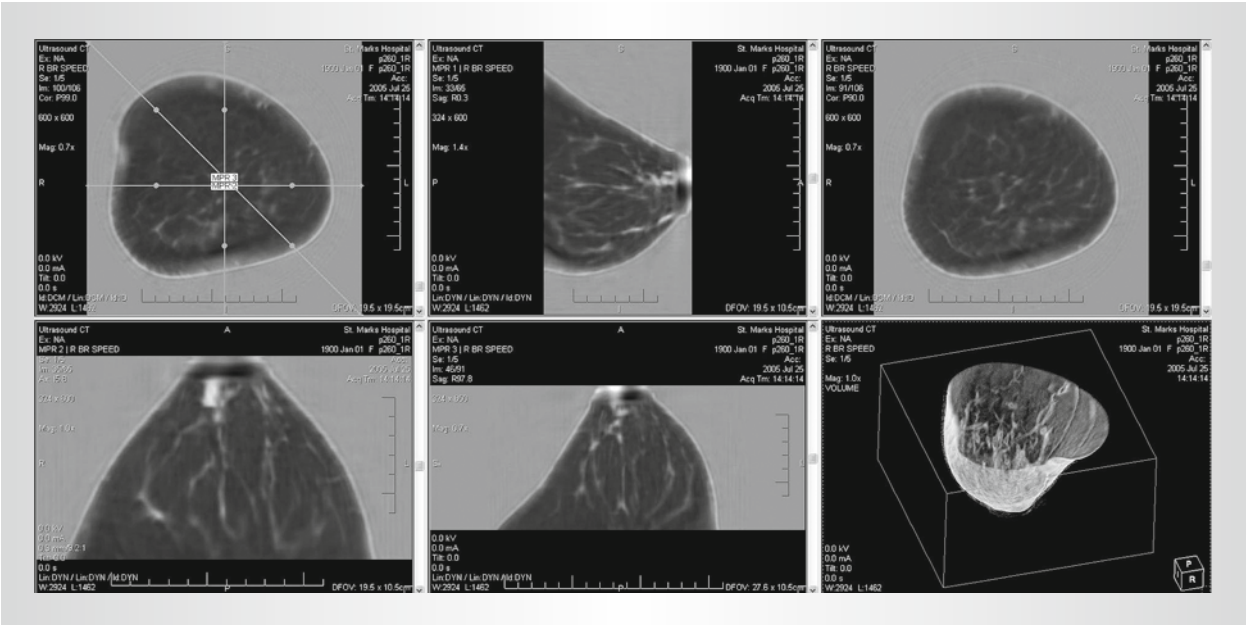

Figure 8. Example scans produced from ultrasound by TechniScan Medical Systems.

Cell processor to meet this goal, but each has disadvantages. FPGAs can be a huge engineering undertaking, and DSPs don't provide the floating-point performance necessary to compute images in minutes. The Cell processor showed potential, but the development workstation was expensive for a small startup.

For approximately $600—the cost of one GeForce 8800 GTX and an ATX power supply—TechniScan was able to start a proof-of-concept project to port the inverse-scattering algorithm to the GPU. The scanner collects ultrasound signals at regular rotational positions and at regular coronal slice positions. The inverse-scattering algorithm is a modified nonlinear conjugate gradient method that tries to match simulated scattered ultrasound to the collected signals. It uses 2D convolution by fast Fourier transform (FFT) to simulate ultrasound propagation.[17] This simulation is used to compute the residual, gradient, and step length for each iteration. Approximately 63 million 2D FFTs are performed during the algorithm's run, which accounts for over 75 percent of the computation time.

A 2D GPU convolution routine can be implemented easily using CUFFT—the Fast Fourier Transform library supplied with CUDA—and a simple kernel to perform point-wise multiplication. This approach is approximately eight times faster than a CPU version using an optimized FFT and running on one core of a 2.4-GHz Core2 Quad Q6600 processor. However, because the FFT size is fairly small (256×64), using the entire GPU to perform a single FFT does not produce enough work to efficiently utilize the GPU. Instead, creating a batched FFT that assigns multiple FFTs to different thread blocks is a much more effective way of utilizing the hardware. Implementing a batched 2D FFT kernel nearly doubled convolution performance and made the GPU convolution almost 16 times faster than the CPU implementation.

Other parts of the algorithm showed more dramatic speed improvements running on the GPU. The most impressive improvements were seen in the CAXPBY routine, which multiplies two complex-valued vectors by two different scalars and sums the resulting vectors; computation of the L2 norm of a complex vector; and computation of the complex exponent of each value of a large, complex vector. Figure 9 shows the runtime of each routine running on a GeForce 8800 GTX and one core of a 2.4-GHz Q6600 GPU.

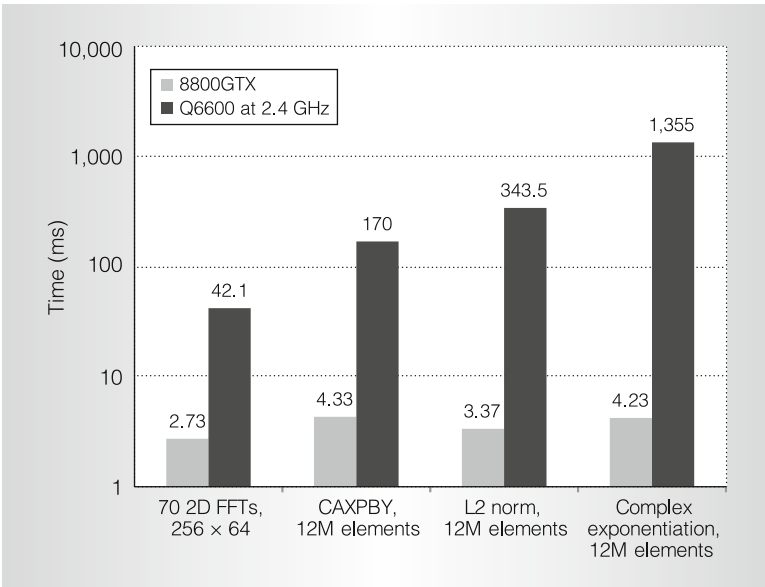

Figure 9. GPU speedup of individual computational routines.

The algorithm was also adapted to run on multiple GPUs. Running the algorithm on two Tesla D870s provided the performance necessary to generate images in approximately 16 minutes—fast enough to meet the same-visit requirement of TechniScan's customers. This level of performance is over twice as fast as a 16-core Intel Core2 CPU cluster.

Fluid dynamics

Physical simulations based on finite-element, finite-difference, finite-volume, and similar methods are not as trivially parallelized as molecular dynamics. However, by adopting a blocking strategy similar to those used in matrix multiplication and image processing, algorithms of this sort can also be transformed into highly parallel computations.

As an example, we consider the 2D compressible Euler equations, which are often used in the design of aerospace vehicle components such as rocket nozzles and supersonic airfoils. The equations are solved on an irregular structured grid using the finite-volume method and an integration scheme developed by Ni.[18] The solver uses local time stepping and multigrid techniques to accelerate convergence. This procedure can serve as the pseudo-time iteration in a more complex solver for the unsteady compressible Navier-Stokes equations with turbulence modeling.

Phillips and colleagues developed this CUDA-based solver,[19] which makes each thread block responsible for updating a 16 × 5 tile of the domain (see Figure 10). Each block requires access to a 20 × 9 area because the computational stencil extends by two nodes past the tile boundary in each direction. These specific dimensions are chosen to best fit Tesla-architecture GPUs: the memory subsystem can deliver much higher bandwidth by coalescing accesses by threads of a warp to 16 contiguous values, and a height of five is the largest that fits within available register and per-block shared memory limits. Memory bandwidth can become the bottleneck when solving the Euler equations on graphics hardware.[20] Consequently, reducing memory bandwidth by calculating intermediate variables on the fly, rather than storing them, can improve efficiency.

Figure 11 shows example simulations of a rocket nozzle and supersonic airfoil performed on a QuadroFX 5600. Figure 12 shows the performance of the CUDA-based solver running on a GPU cluster in comparison to a serial reference solver running on a 2.4-GHz Core2 Duo. The cluster consists of four nodes, each with two QuadroFX 5600 GPUs and dual Opteron 2216 CPUs connected by gigabit Ethernet. For the solution process, the domain is decomposed across GPUs, and each GPU performs one iteration of the solver, after which the boundary elements are communicated with its neighbors. On the coarsest grid of 1,600 nodes, the amount of parallel work is small enough that the overhead of moving work onto the GPU is a comparatively high cost, and a single GPU is only able to deliver about four times the performance of the serial CPU solver. With 25,000 nodes, the subdomains remain small and communication time dominates; a single GPU, which solves the problem at roughly 18 times the speed of the CPU, is faster than the entire eight GPU cluster. As the grids become denser, communication cost is an ever-decreasing component of the solution time. At the densest grid resolu-

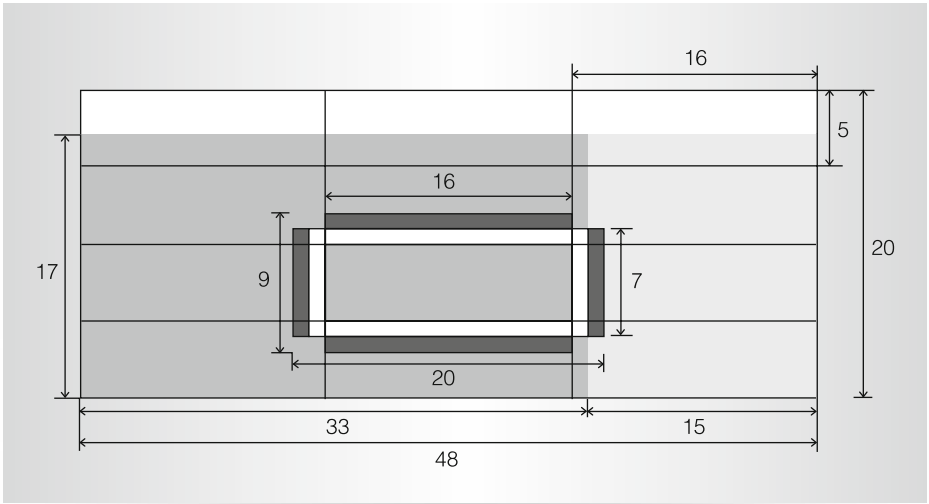

Figure 10. Decomposition of a 33 × 17 subdomain. A single-thread block computes a result for the light gray tile in the center and also accesses neighboring information for integration (white boxes surrounding the center tile) and smoothing (dark gray boxes surrounding the center tile) overlap.

tion, the solver scales nearly linearly as more GPUs are used, delivering 22 times the serial performance on one GPU and 160 times the serial performance on eight GPUs. All GPU simulations were performed in single precision, and the results were verified to agree with a reference CPU implementation. The use of single precision required a priori calculation of the inlet and exit properties to compensate for numerical drift due to the exponentiation and square-root functions' limited accuracy.

Seismic imaging

The petroleum industry makes heavy use of seismic data to construct images of the Earth's subsurface structure in its search for oil and gas. A seismic survey of a prospective region will typically consist of hundreds of thousands of seismic experiments. Each experiment involves an impulsive acoustic source that generates a signal that propagates up to tens of kilometers through the subsurface, reflects off the interfaces between rock layers, and is recorded by a few thousand pressure-sensitive receivers at the surface. This acquisition process produces terabytes of seismic data.

Reconstructing a subsurface image from this data typically requires weeks of computations on thousands of CPUs, with the exact effort depending on the fidelity of the approximations used to simulate the wave propagation.

Given the large amount of parallel computation involved in the seismic imaging process, it is natural to consider mapping the process onto the GPU with CUDA. We consider a specific imaging method, where the wave equation is solved in the frequency domain using a paraxial approximation implemented with an ADI (alternating-direction implicit) finite-difference method. This method's computational cost is dominated by the evaluation and

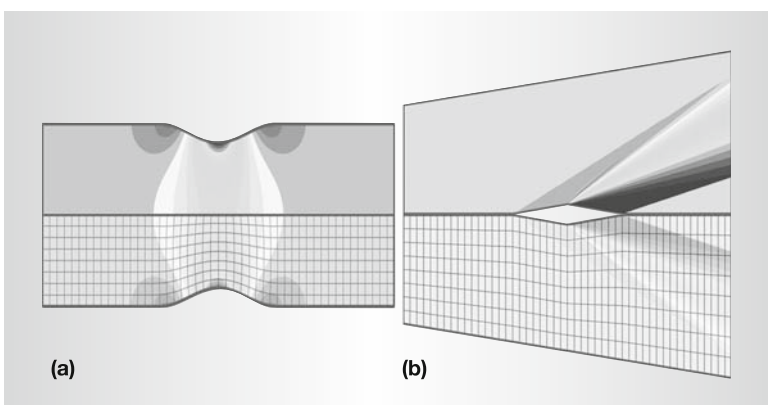

Figure 11. Simulation results showing pressure distribution in a rocket nozzle (a) and over a supersonic airfoil (b).

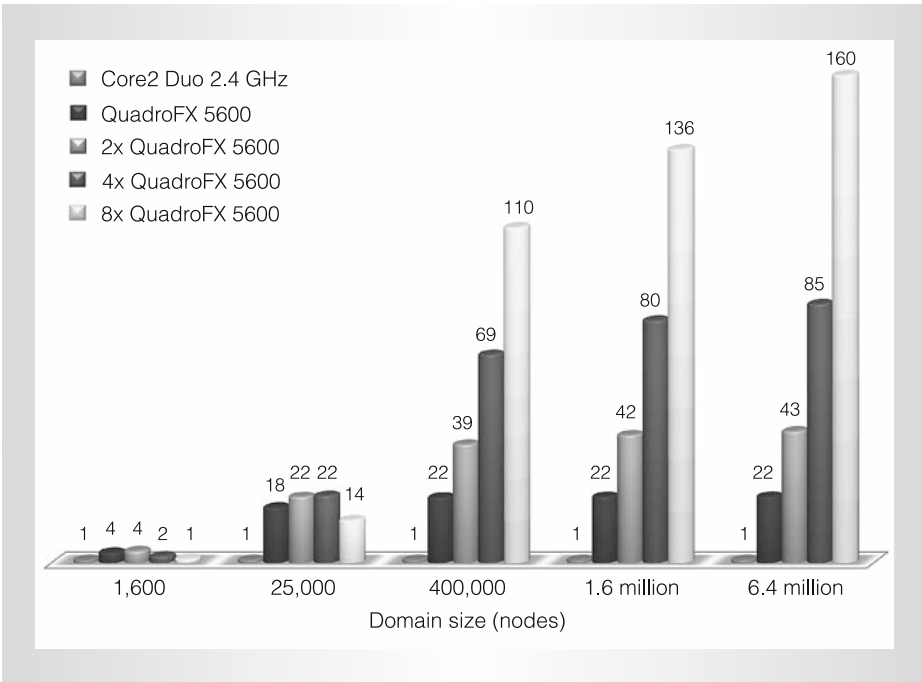

Figure 12. Performance speedup for GPU clusters of varying size over a single CPU for solving 2D Euler equations.

solution of complex-valued tridiagonal linear systems, where there are on the order of a billion systems for each seismic experiment and the dimension of each system is in the hundreds.

Although the solution of tridiagonal systems can be parallelized directly,[21,22] this is not an effective way to solve such a large number of relatively small systems. A much more efficient approach is to assign each linear system to a single thread. In this particular application, it also happens that many systems to be solved involve the same coefficient matrix. This leads to a natural design in CUDA where each thread block is given one tridiagonal matrix and each thread of that block solves a linear system involving that matrix and a specific right-hand side. Furthermore, keeping these shared coefficients in shared memory avoids redundant accesses to memory. A thread block's threads collectively read their coefficients into shared memory and synchronize, and then each thread computes its system's solution.

Figure 13 examines the speedup of the prototype CUDA implementation over CPU production code with varying numbers of processors. The CUDA code was executed on a Tesla C870 GPU. CPU performance was measured on two systems, a dual 3.6-GHz Intel Xeon and a dual 3.0-GHz quad-core Xeon (Harpertown). The CPU code was optimized to use SSE (streaming SIMD extension) vector instructions and compiled with the Intel Fortran compiler. Even though eight CPU cores were available on the Harpertown-based system, performance did not scale beyond four processes because of memory bandwidth limitations. This resulted in a single GPU out-performing an entire Harpertown system by a factor of six on this seismic imaging code.

In considering the experiences of a number of applications parallelized with CUDA, we believe that a few basic trends are apparent. CUDA provides a straightforward means of describing inherently parallel computations and is particularly well suited for data-parallel algorithms. The NVIDIA Tesla GPU architecture delivers high computational throughput on such kernels. Furthermore, unlike the massively parallel machines of the past, which were large,

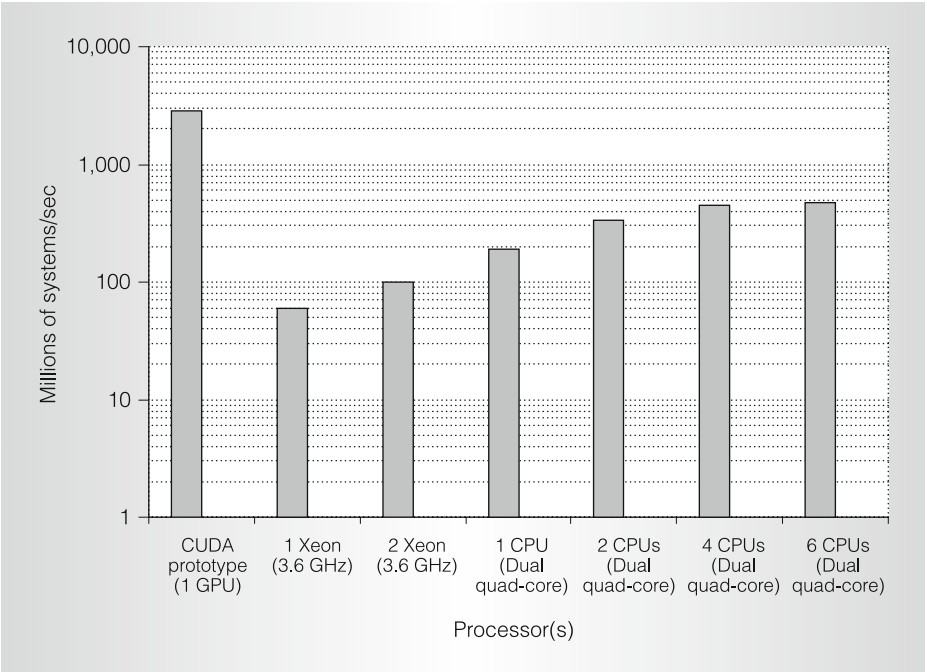

Figure 13. Speedup of a CUDA prototype wave-equation solver compared with various CPU configurations.

expensive, and available to a select few, these GPUs capable of running CUDA are ubiquitous. By the end of summer 2008, NVIDIA will have shipped roughly 80 million CUDA-capable GPUs, transforming acceleration with massively parallel hardware from a rarity into an everyday commodity.

Reviewing the many CUDA-enabled applications now available, we encounter a few important design techniques. First, and foremost, is the fundamental importance of exposing sufficient amounts of fine-grained parallelism to exploit hardware like the Tesla-architecture GPU. Second is the importance of blocking computations, a process that naturally fits the CUDA thread block abstraction and encourages data layout and access patterns with high locality. Third is the efficiency of data-parallel programs where threads of a warp follow the same execution path, thus fully utilizing the GPU's processor cores. Finally is the benefit of the on-chip, per-block shared memory provided by the Tesla architecture, which provides high-speed, low-latency scratchpad space that is critical to the performance of many efficient algorithms.

The latest CUDA toolkit, documentation, and code examples, as well as a directory of some of the many available CUDA-based applications and research projects, are available at www.nvidia.com/CUDA/. A course on parallel programming using CUDA is also available online (http://courses.ece.uiuc.edu/ece498/al). MICRO

References

1. J. Nickolls et al., ''Scalable Parallel Programming with CUDA,'' *ACM Queue*, vol. 6, no. 2, Mar./Apr. 2008, pp. 40-53.
2. E. Lindholm et al., ''NVIDIA Tesla: A Unified Graphics and Computing Architecture,'' *IEEE Micro*, vol. 28, no. 2, Mar./Apr. 2008, pp. 39-55.
3. B. Catanzaro, N. Sundaram, and K. Keutzer, ''Fast Support Vector Machine Training and Classification on Graphics Processors,'' *Proc. 25th Ann. Int'l Conf. Machine Learning*, Omnipress, 2008, pp. 104-111.
4. B. He et al., ''Relational Joins on Graphics Processors,'' *Proc. ACM SIGMOD 2008*, ACM Press, 2008; www.cse.ust.hk/catalac/papers/gpujoin_sigmod08.pdf.

5. M. Schatz et al., ''High-Throughput Sequence Alignment Using Graphics Processing Units,'' *BMC Bioinformatics*, vol. 8, no. 1, 2007, p. 474; http://dx.doi.org/10.1186/1471-2105-8-474.
6. S. Manavski and G. Valle, ''CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment,'' *BMC Bioinformatics*, vol. 9, suppl. 2, 2008, p. S10; http://dx.doi.org/10.1186/1471-2105-9-S2-S10.
7. S.S. Stone et al., ''How GPUs Can Improve the Quality of Magnetic Resonance Imaging,'' *Proc. 1st Workshop General Purpose Processing on Graphics Processing Units,* 2007.
8. D. Frenkel and B. Smit, *Understanding Molecular Simulations*, Academic Press, 2002.
9. J.A. Anderson, C.D. Lorenz, and A. Travesset, ''Micellar Crystals in Solution from Molecular Dynamics Simulations,'' *J. Chemical Physics*, vol. 128, 2008, pp. 184906-184916.
10. J.A. Anderson, C.D. Lorenz, and A. Travesset, ''General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units,'' *J. Computational Physics*, vol. 227, no. 10, May 2008, pp. 5342-5359.
11. J.E. Stone et al., ''Accelerating Molecular Modeling Applications with Graphics Processors,'' *J. Computational Chemistry*, vol. 28, no. 16, 2007, pp. 2618-2640.
12. C.I. Rodrigues et al., ''GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications,'' *Proc. 2008 Conf. Computing Frontiers* (CF 08), ACM Press, 2008, pp. 273-282.
13. S. Plimpton, ''Fast Parallel Algorithms for Short-Range Molecular Dynamics,'' *J. Computational Physics*, vol. 117, no. 1, 1995, pp. 1-19.
14. M. Shirts and V.S. Pande, ''Screen Savers of The World Unite,'' *Science*, vol. 290, no. 5498, 2000, pp. 1903-1904.
15. V. Volkov and J.W. Demmel, ''LU, QR and Cholesky Factorizations Using Vector Capabilities of GPUs,'' tech. report UCB/EECS-2008-49, EECS Dept., Univ. of Calif., Berkeley, 2008.
16. S. Ryoo et al., ''Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA,'' *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, ACM Press, 2008, pp. 73-82.
17. S.A. Johnson et al., *Apparatus and Method for Imaging Objects with Wavefields*, US patent 6,636,584, Patent and Trademark Office, 2003.
18. R.H. Ni, ''A Multiple Grid Scheme for Solving the Euler Equations,'' *Proc. AIAA 5th Computational Fluid Dynamics Conf.*, AIAA Press, 1981, pp. 257-264.
19. E.H. Phillips et al., ''A Multi-Grid Solver for the 2D Compressible Euler Equations on a GPU Cluster,'' tech. report ECE-CE-2008-2, Computer Eng. Research Lab., Univ. of California, Davis, 2008; www.ece.ucdavis.edu/cerl/techreports/2008-2.
20. T. Brandvik and G. Pullan, ''Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware,'' *Proc. 48th AIAA Aerospace Sciences Meeting and Exhibit*, AIAA Press, 2008, p. 607.
21. H.S. Stone, ''Parallel Tridiagonal Equation Solvers,'' *ACM Trans. Mathematical Software*, vol. 1, no. 4, Dec. 1975, pp. 289-307, http://doi.acm.org/10.1145/355656.355657.
22. M. Kass, A. Lefohn, and J. Owens, ''Interactive Depth of Field Using Simulated Diffusion on a GPU,'' tech. report 06-01, Pixar Animation Studios, 2006; http://graphics.pixar.com/DepthOfField/.

Michael Garland is a research scientist at NVIDIA. His research interests include computer graphics and visualization, geometric algorithms, and parallel algorithms and programming models. Garland received his PhD in computer science from Carnegie Mellon University.

Scott Le Grand is a senior engineer on the CUDA software team at NVIDIA. Le Grand received a BS in biology from Siena College and a PhD in biochemistry from Pennsylvania State University.

John Nickolls is director of architecture at NVIDIA for GPU computing. His interests include parallel processing systems, languages, and architectures. Nickolls received his

PhD in electrical engineering from Stanford University.

Joshua Anderson is a graduate student in the Department of Physics and Astronomy at Iowa State University and the Ames Laboratory. His PhD work focuses on the design and implementation of a general-purpose molecular dynamics simulation code on the GPU and the theoretical design of new polymeric materials by computational methods. Anderson received his BS in physics and computer science from Michigan Tech.

Jim Hardwick is the senior software engineer at TechniScan Medical Systems in Salt Lake City, Utah. His expertise is in medical device development, including software and hardware development and human-computer interaction. Hardwick received his BS in computer science from Westminster College.

Scott Morton leads a geophysical technology group at Hess Corporation. His research interests include geophysical technologies with a focus on seismic imaging. Morton received his PhD in astrophysics from the University of Illinois at Urbana-Champaign.

Everett Phillips is a graduate student at the University of California, Davis, working in mechanical and aeronautical engineering and electrical and computer engineering. His research interests include computational fluid dynamics, parallel computing, and graphics hardware. Phillips received his BS in mechanical engineering from the University of California, Davis.

Yao Zhang is a PhD student in the Department of Electrical and Computer Engineering at University of California, Davis. Zhang received his BS in electrical engineering from the Beijing Institute of Technology. His research interests are in the area of GPU computing, especially in solving linear algebra and fluid problems on GPUs.

Vasily Volkov is a PhD student in the Department of Electrical Engineering and Computer Science at the University of California, Berkeley. His reserach interests include high-performance computing and numerical linear algebra. Volkov received his MS from the Moscow Institute of Physics and Technology and MEng from the Nanyang Technological University.

Direct questions and comments about this article to Michael Garland, NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050; mgarland@nvidia.com.