Computational Fluid Dynamics Simulations Using Many Graphics Processors

In this scenario, computational fluid dynamics simulations of turbulence are performed with 64 GPUs and an optimized CFD algorithm using communication/computation overlapping. Detailed timings reveal that the GPUs' internal calculations are so efficient that operations related to data exchange between compute nodes now cause a scaling bottleneck on all but the largest problems.

Computational fluid dynamics (CFD) is a reasonable prototype for many of the algorithms used in computational science today. The solution of the Navier-Stokes equations for the evolution of fluid flow is reasonably complex numerically and involves solving a nonlinear, coupled system of hyperbolic and elliptic partial differential equations (PDEs). At the algorithm level, CFD is fundamentally quite similar to computational electromagnetics (Maxwell's equations), computational physics (Schrödinger's equation), computational biology (protein folding), computational solid mechanics (using finite-element methods), and many other applications in science and engineering. The Navier-Stokes equation for incompressible flow is

$$\nabla \cdot u = 0$$
$$\rho\left(\frac{\partial u}{\partial t} + u \cdot \nabla u\right) = -\nabla p + \mu \nabla^2 u.$$

For turbulence simulation, the large range of spatial scales means these equations need to be discretized on very large meshes involving an order of many billions of unknowns. Therefore, both memory and time constraints force the solution to be computed in parallel on many processors. This article is concerned with the problems inherent in massively parallel GPU calculations, because the speed of GPU processors moves GPU supercomputers into a new performance regime. In this article, our focus is on GPUs as the processing hardware, and we present an algorithm to optimize both the local GPU performance and inter-GPU performance. For some background information on this topic, see the sidebar, "Related Work in Computational Fluid Dynamics with GPUs."

Implementation

In our implementation, the single-instruction, multiple-data (SIMD) core count (which is normally 240 cores on a single GPU) is increased to 15,360 (GPU cores) by using many GPUs together. The GPUs communicate via their CPU hosts and a message passing interface (MPI). Next we describe the GPU supercomputer hardware used for these simulations.

1521-9615/12/$31.00 © 2012 IEEE
Copublished by the IEEE CS and the AIP

Ali Khajeh-Saeed and J. Blair Perot
University of Massachusetts, Amherst

Related Work in Computational Fluid Dynamics with GPUs

GPUs typically have 32 single-instruction, multiple-data (SIMD) cores on a multiprocessor, and many multiprocessors on a GPU chip (16 on the Fermi). In our work, the core count is further increased to 15,360 GPU cores by using many GPUs together, with the GPUs communicating via their CPU hosts and the message passing interface (MPI) protocol. GPUs have the advantage of high memory bandwidths (178 Gbytes for the M2090). This makes them attractive for memory-bound algorithms. Examples of such algorithms include bioinformatics,[1] graph theory,[2] and partial differential equation (PDE) solutions.[3] GPUs have the disadvantage of consuming more power than CPUs, and being more difficult to program to achieve high efficiency.

Early examples of CFD calculations on GPUs used OpenGL processing and were very different because the GPU hardware itself has changed radically over the last five years. Erich Elsen and his colleagues[4] and Andrew Corrigan and his colleagues[5] discuss more recent implementations that involve the CUDA programming paradigm. Paulius Micikevicius[6] applied a 3D finite-difference computation using CUDA on four GPUs and achieved linear speedup for up to four GPUs. Diego Rossinelli and his colleagues[7] described a 2D simulation of a bluff body using a vortex particle method on the GPU that achieved a speedup of 25. Inanc Senocak and his colleagues[8] discretized the Navier-Stokes equations on a uniform Cartesian staggered grid with a second-order central difference scheme. They achieved an 11-times speedup compared to an eight-core CPU (using OpenMP) for a single GPU and a 130-times speedup for 128 GPUs (versus an eight-core CPU). Wei Ran and his colleagues[9] implemented a 1D space-time conservation-element and solution-element (CESE) method and applied this to shock-tube problems. They achieved a maximum of 71-times speedup with a 9800 GT compared to the single core of the Intel E7300 CPU.

Today, several single GPU codes for computing fluid flow are based on Jos Stam's stable scheme.[10] These codes are excellent for visual purposes but aren't sufficiently accurate for physical simulations.

References

1. A. Khajeh-Saeed, S. Poole, and J.B. Perot, "Acceleration of the Smith-Waterman Algorithm Using Single and Multiple Graphics Processors," *J. Computational Physics*, vol. 229, no. 11, 2010, pp. 4247–4258.
2. T. McGuiness and J.B. Perot, "Parallel Graph Analysis and Adaptive Meshing Using Graphics Processing Units," *Proc. 18th Ann. Conf. CFD Soc. Canada*, CFD Soc. Canada, 2010; www.ecs.umass.edu/mie/tcfd/ConferencePapers/GPU_Graph_Analysis.pdf.
3. J. Krüger and R. Westermann, "A GPU Framework for Solving Systems of Linear Equations," *GPU Gems 2*, Addison-Wesley, 2005, pp. 703–718.
4. E. Elsen, P. LeGresley, and E. Darve, "Large Calculation of the Flow over a Hypersonic Vehicle Using a GPU," *J. Computational Physics*, vol. 227, no. 4, 2008, pp. 10148–10161.
5. A. Corrigan et al., "Running Unstructured Grid-Based CFD Solvers on Modern Graphics Hardware," *Int'l J. Numerical Methods in Fluids*, vol. 66, no. 2, 2011, pp. 221–229.
6. P. Micikevicius, "3D Finite Difference Computation on GPUs Using CUDA," *Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ACM, 2009, pp. 79–84.
7. D. Rossinelli et al., "GPU Accelerated Simulations of Bluff Body Flows Using Vortex Particle Methods," *J. Computational Physics*, vol. 229, no. 9, 2010, pp. 3316–3333.
8. D.A. Jacobsen, J.C. Thibault, and I. Senocak, "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," *Proc. 48th AIAA Aerospace Sciences*, American Inst. Aeronautics and Astronautics (AIAA), 2010; http://works.bepress.com/inanc_senocak/3.
9. W. Ran et al., "GPU Accelerated CESE Method for 1D Shock Tube Problems," *J. Computational Physics*, vol. 230, no. 24, 2011, pp. 8797–8812.
10. J. Stam, "A Simple Fluid Solver Based on the FFT," *J. Graphics Tools*, vol. 6, no. 2, 2001, pp. 43–52.

Hardware

Our work was primarily computed on the Lincoln supercomputer housed at the US National Center for Supercomputing Applications (NCSA). This machine has 96 Tesla S1070s (384 GPUs total). Each GPU has 4 Gbytes of memory and a theoretical bandwidth of 102 Gbytes per second. Also, each GPU has a PCI-Express 4× connection (2 Gbytes per second) to its CPU host. We also performed tests on Orion, which is an in-house GPU machine containing a GTX 480, a Tesla C2070, and two GTX 295 cards. We're interested here in comparing the GPU results to the CPU cores on Lincoln (a quad-core Intel 64 Harpertown).

Our low-level CFD algorithm's structure is dictated by two key features of the GPU hardware. First, the GPU's read/write memory is one order of magnitude faster when the memory is read linearly. Second, each GPU multiprocessor has fast on-chip memory (shared memory), which serves essentially as an addressable program-supervised cache. CFD requires considerable random memory accesses (even when using structured meshes).

Roughly 90 percent of these slow random memory accesses can be eliminated by

- reading large chunks of data into the shared-memory space linearly, which is fast for all accesses;
- operating on the data in the shared memory; and
- writing the processed data back to the main GPU memory (global memory) linearly.

This optimization is the key to speeding up the GPU 45 times more than a CPU.

Software

Our solution method uses a three-step, low-storage Runge-Kutta scheme for time advancement that's second-order accurate (in space).[1] This scheme is stable for eigenvalues on the imaginary axis less than 2, which implies $CFL < 2$ for advective stability (where CFL is the Courant-Friedrichs-Lewy number). Our simulations always use a maximum $CFL < 1$. The diffusive terms are advanced with the trapezoidal method for each Runge-Kutta substep, and the pressure is solved using a classical fully discrete fractional step method,[2] although an exact fractional step method is also possible.[3] The solution of the elliptic pressure Poisson equation dominates the solution time (roughly 90 percent), so we present further details of this portion of the code in the next section. We also present timing results later that focus on this part of the algorithm.

For spatial discretization, we use a second-order Cartesian staggered mesh scheme. This not only conserves mass and momentum to machine precision, but because it's a type of discrete calculus, this scheme also conserves vorticity (or circulation) and kinetic energy in the absence of viscosity.[4] As a result, there's no artificial viscosity or diffusion in this method except for that induced by the time-stepping scheme.[5] In addition, the staggered mesh discretization is free from pressure modes and the need for pressure stabilization terms. This scheme's physical fidelity makes it highly appropriate for direct numerical simulations of turbulence in geometries with walls (we provide a summary of references elsewhere[6]). A Cartesian mesh method is also appropriate for most large simulations of turbulence. The goal of fluid simulations on supercomputers that use many GPUs is typically to study turbulence, not complex geometries. Nevertheless, the code is currently being converted to use an unstructured mesh, and in future work we'll discuss this aspect of the implementation.

We performed our CFD simulations on 512^3 meshes (with roughly half a billion unknowns) with fully periodic boundary conditions on the exterior of the computational domain, and wall-boundary conditions on interior embedded objects. We provide further simulation details elsewhere.[7]

Partitioning

All PDE solution methods ultimately involve placing numerous unknowns (that approximate the solution) into a 3D domain and evolving those unknowns in time. Parallel-solution algorithms typically partition these unknowns spatially among the available processing units in groups called *subdomains*.

To calculate a coupled solution, subdomains must invariably communicate data with their neighbors. In these simulations, the amount of subdomain communication is minimized by choosing subdomains with minimal surface area. Only data on the boundary, or surface, of the subdomain is communicated, and communication occurs only with the six local neighboring subdomains. For example, for a 512^3 simulation running on 64 GPUs, each subdomain is 128^3 and each GPU communicates $6 \times 128^2 = 0.1$ million data items (or approximately 5 percent of its data). For optimal scaling, the MPI must be "hidden" by overlapping communication with useful computations.

For an efficient GPU solution, the subdomains (one each per GPU) must also be further partitioned into *chunks* (what Nvidia calls *blocks*). On the GPU, each multiprocessor will handle one or more chunks of data. In our implementation, each chunk is 16 mesh points in x (or a multiple of 16), any dimension in the y direction (though we typically chose a multiple of 16 here), and the full subdomain extent in the z direction. For efficient processing, it's best to have at least two chunks per multiprocessor. The GPU multiprocessors can hide memory latencies by having two chunks active at the same time. The size of 16 in the x direction is dictated by the fact that the GPU multiprocessors each have 32 SIMD cores that read data much more quickly ("coalesced memory access" in Nvidia terminology) if they can do it in groups of 16. We can construct subdomains and chunks for unstructured meshes using mesh-partitioning software such as Metis (see http://glaros.dtc.umn.edu/gkhome/views/metis).

All CFD algorithms are strongly memory bound. The memory speeds are therefore critical to the code's efficiency and scaling. The key issue with multi-GPU computing is that there are a minimum of three widely differing memory speeds. On a GPU, a chunk of data typically is read into and operated on using fast shared memory (which is effectively a cache). The amount of shared memory is fixed by the hardware and is 8 (Tesla S1070) or 24 Kbytes per chunk when running two chunks per multiprocessor on the Fermi GPUs. A chunk's boundary data is accessed via an order-of-magnitude slower random-memory reads to the GPU's main memory. Subdomain boundary data must also be transferred, and is accessed via MPI calls that are yet another order of magnitude slower. Although our CFD implementation aggressively minimizes the number of slower memory accesses, these boundary communications still impact the code performance on everything but the largest problem sizes.

Optimizations

Memory allocation and deallocation are expensive on the GPU (and the CPU), because they require calls to the operating system. To avoid this, the code sets up its own temporary arrays at start up. Any subroutine can hold and drop these quickly. They're only deallocated when the code completes.

Grid information such as Δx, Δy, Δz are stored in a special GPU memory space called constant memory. Constant memory is cached, and it's therefore fast for repeatedly called items, such as geometry parameters.

Data transfers between the CPU and GPU are relatively slow and expensive (5 to 10 Gbytes/s). Almost all data for the CFD calculation therefore resides on the GPU and stays on the GPU. We only copy data to the CPU when it's needed (for MPI, or to write data to files).

We wrote the code so that low-level operations come in either a CPU or GPU version. The user makes a choice at compile time, and then the compiler optimizes the code for that given hardware.

Algorithm Details

We achieved considerable efficiency through our optimized algorithm. Here we detail how it was implemented.

GPU Implementation

The code solves the pressure Poisson equation using a polynomial preconditioned conjugate gradient (PPCG) iterative method. The PPCG method is an efficient iterative method and is guaranteed to converge for a symmetric, positive-definite matrix. The method is composed of one large sparse-matrix multiply ($\boldsymbol{w} = \boldsymbol{M}\boldsymbol{p}$), one large sparse-preconditioner-matrix multiply ($\boldsymbol{z} = \boldsymbol{P}\boldsymbol{r}$), two scalar (dot) products ($\alpha = \mathbf{z} \times \mathbf{r}$ and $\gamma = \mathbf{p} \cdot \mathbf{w}$), and three alpha X plus Y (AXPY) operations ($\boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{w}$, $\boldsymbol{x} = \boldsymbol{x} + \alpha\boldsymbol{p}$, $\boldsymbol{p} = \boldsymbol{z} + \beta\boldsymbol{p}$). The three AXPY parts are easily mapped to the GPU architecture. However, the most computationally intensive part of the solution procedure is the sparse-matrix multiplies that compute the Cartesian-mesh discrete Laplacian ($\boldsymbol{M}$) and its approximate inverse ($\boldsymbol{P}$). In the current implementation, the preconditioner has the same sparsity pattern as the Laplacian matrix, and it's therefore implemented in exactly the same way as the Laplacian. To compute the Laplacian matrix for a particular cell, all neighboring cells and the central cell are needed (seven cells in 3D). The pseudocode for the operator $\boldsymbol{M}$ is

$$\begin{aligned} w[i,j,k] = {} & p[i,j,k] \times \mathrm{diag}[i,j,k] + (p[i+1,j,k] \\ & + p[i-1,j,k]) \times \Delta x^2[i] + (p[i+1,j,k] \\ & + p[i,j-1,k]) \times \Delta y^2[j] + (p[i,j,k+1] \\ & + p[i,j,k-1]) \times \Delta z^2[k]. \end{aligned}$$

When performed naively, the seven-point matrix stencil reads each data item seven times from the main GPU memory. Only the first three values are linear, stride-one memory accesses and therefore fast; the others are large-stride memory accesses and essentially random memory operations. We made the code more efficient using a modified version of Paulius Micikevicius' implementation.[8] This involves reading the data once into the shared memory on each GPU multiprocessor, and then accessing it from this fast-memory location seven times. To do this, each multiprocessor keeps three *XY* planes of data (from the data chunk) in its memory. The middle *XY* plane contains five of the stencil points (in the *X* and *Y* directions) saved in shared memory (in the *ps* temporary variable that we discuss more in a bit), while the upper and lower *XY* planes contain the sixth and seventh stencil values (just above and below the middle *XY* plane) saved in registers. After the discrete Laplacian is computed for the middle plane, the middle (shared memory) and upper (register memory) planes are copied to the lower (register memory) and middle (shared memory) planes, respectively. The upper plane then reads in the new data from the main (global) GPU memory to the register memory. In pseudocode, **ps** is now a planar array in fast-shared memory, and p_{upper} and p_{lower} are in registers:

$$\begin{aligned} w[i,j,k] = {} & \mathbf{ps}[i,j] \times \mathrm{diag}[i,j,k] + (\mathbf{ps}[i+1,j] \\ & + \mathbf{ps}[i-1,j]) \times \Delta x^2[i] + (\mathbf{ps}[i,j+1] \\ & + \mathbf{ps}[i,j-1]) \times \Delta y^2[j] + (p_{\mathrm{upper}} + p_{\mathrm{lower}}) \\ & \times \Delta z^2[k]. \end{aligned}$$

After this is computed for every i, j value in the chunk, the planes are shifted up and new data is read-in (for p_{upper}) only. This scheme requires reading only one input value for every output value that's computed (rather than seven). It can be adapted to unstructured meshes fairly easily using explicit neighbor pointers instead of $[i, j, k]$ location indexing.

However, to compute a $16 \times 16 \times NZ$ chunk of data (where NZ is the chunk size in the z-direction), an 18×18 data plane of input is required (minus the four corners). This is read in as a 16×18 block (with stride-one fast access), and two 1×16 strips for the two sides. These last two strips have a stride equal to the subdomain's size in the x-direction, and therefore are much slower to read. It therefore takes roughly the same amount of time to read the two 1×16 strips as it does the rest of the data (18×16). This is the first example where the internal GPU data is processed so efficiently that the unusual operations (two boundary strips in this case) take just as much time.

The other option would be to read 16×16 blocks efficiently (with no side strips) but only compute a 14×14 region of the stencil. Because the SIMD cores process 16 items at a time, this means that the code would have an instruction divergence—that is, some cores would perform an operation, while other cores do something else. On these SIMD cores, this results in slower execution (by about a factor of two). In addition, this approach means the multiprocessors are reading blocks of 16 that overlap at the edges. This makes the 16×16 read slower. Therefore, the advantage of reading no boundary strips is actually lost.

The second major optimization in the PPCG algorithm is to perform the two dot products at the same time as the matrix multiply and preconditioner-matrix multiply (one dot product along with each matrix). Both arrays for the dot product are already in fast-shared memory when performing the matrix multiply, so this saves reading the two arrays for each dot product (four array reads total). The dot products are therefore essentially free of any time impact on the code, except that their final result must be summed among all of the GPUs. This requires an MPI all-to-all communication that can't be hidden by any useful computations (but the amount of data communicated is small—one word per GPU). It's possible to restructure the PPCG algorithm so that it overlaps dot-product summations with computation; however, this also leads to a PPCG algorithm with more storage and more memory read/writes. In any case, we wouldn't expect the modified PPCG algorithms' speed improvement to be significant.

Finally, with 512^3 meshes, naive summation (for turbulence averages) can lead to round-off errors on the order of 10^9 times the machine precision (for single precision this would mean a first-order error). Although we use double precision in all computations, we still perform the summation in stages to reduce the round-off error. The 3D array is first collapsed into a 2D array using the GPU by summing along the Z direction. Further reduction is then performed in the Y direction, then the X direction on the CPU, and then by summing the results from all the GPUs using MPI all-to-all communication (four stages in total). This procedure only loses roughly two decimal places of accuracy during the summation, and allows the computation's expensive portion (the first reduction to XY planes) to be performed on the fast GPUs.

Multi-GPU Implementation

The approach to parallelism when using many GPUs together is quite different from the type of parallelism used within each GPU. The key aspect of the inter-GPU algorithm is the relatively long communication times (using MPI) between GPU subdomains. These long times are caused by GPU-to-CPU copy times and CPU-to-CPU MPI communication times. For a transfer, all data must be copied from the GPU to the CPU over the PCI Express bus. Only then can the CPU core use MPI (or CPU threads) to communicate the data. The MPI (or thread) communication then occurs at CPU memory speeds (which are slower than GPU main-memory speeds). The Lincoln supercomputer has an InfiniBand single-data rate (SDR) serial link. Orion (our in-house machine) uses MPI on shared memory, which is as fast as MPI can theoretically function (and is about 8 percent slower than using threads directly). After the MPI calls, the data must be copied back to the GPU.

To hide the copy time and the slow MPI communication times, data is prefetched and overlapped with GPU computations as much as possible. A subroutine's basic structure (see Figure 1) is therefore as follows:

- *Step A*. On the GPU, load the six boundary planes of the subdomain data (which resides in the GPU's main memory) into six smaller (stride-one) arrays. This requires the GPU and usually can't be overlapped well with GPU computations.

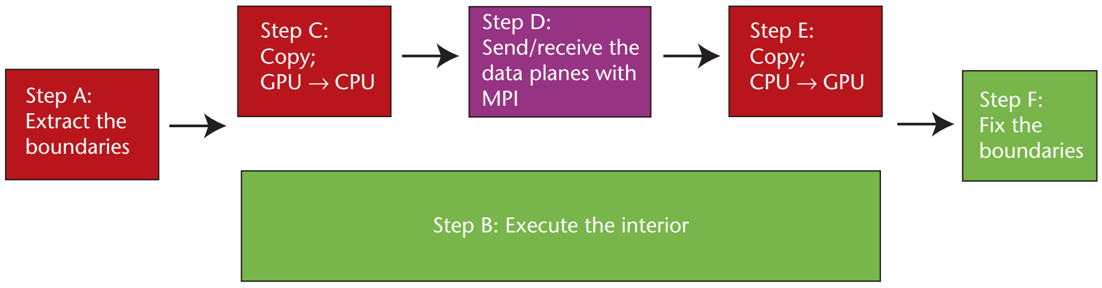
Step A: Extract the boundaries
Step B: Execute the interior
Step C: Copy; GPU → CPU
Step D: Send/receive the data planes with MPI
Step E: Copy; CPU → GPU
Step F: Fix the boundaries

Figure 1. Typical flowchart for subdomain processing. Red indicates boundary operations (running on stream 2), green indicates the primary bulk operation (running on stream 1), and purple indicates message passing interface (MPI) send/receive instructions (running on the CPU).

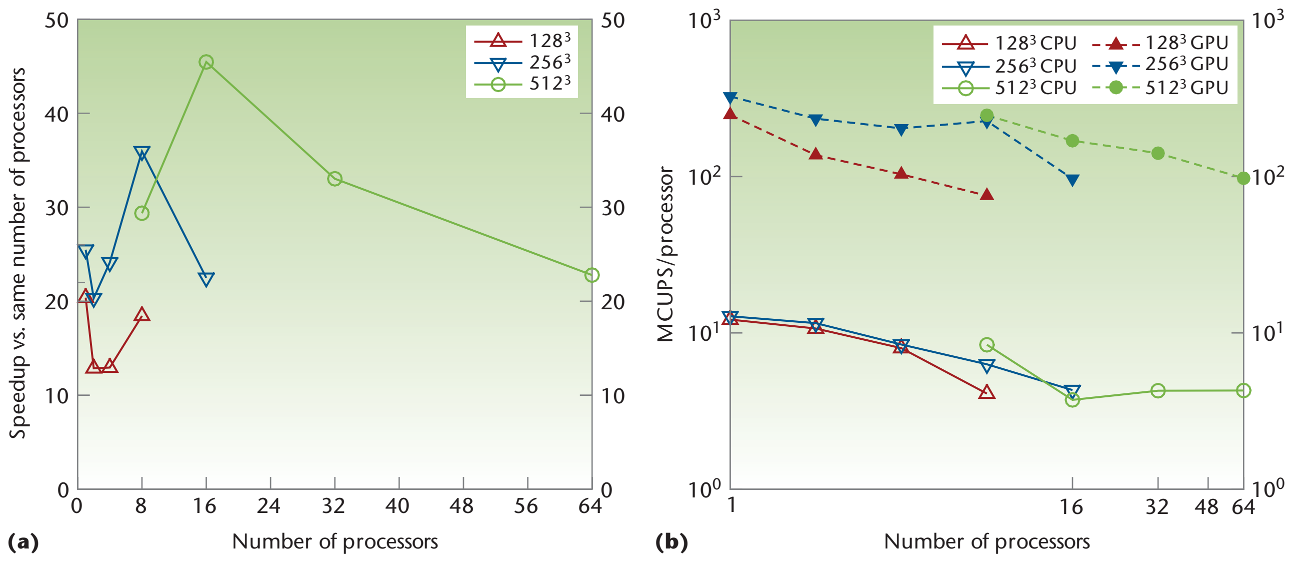

Figure 2. Strong scaling results. (a) Speedup and (b) performance per processor for strong scaling of the 128^3, 256^3, and 512^3 computational fluid dynamics (CFD) problem on Lincoln using GPUs and CPUs. MCUPS stands for millions of cell updates per second.

- *Step B*. On the GPU, start the internal calculation. This step is the subroutine's primary action.
- *Step C*. Copy the small boundary arrays from step A to the CPU. This can overlap with step B.
- *Step D*. When step C is finished, send/receive the data planes using MPI. The CPU handles all MPI operations and is otherwise idle, so this can also overlap with step B.
- *Step E*. Copy the received data from the CPU back to the GPU. Again, this still can overlap with step B.
- *Step F*. When both steps B and E are finished, apply the boundary data to the calculation.

On the latest GPU architecture (Fermi), it's possible to run up to 16 kernels at the same time. So in theory, we can execute steps A and B at the same time if there are available GPU resources. In practice, the code rarely goes faster when doing this. If the internal calculation (step B) takes long enough, it can hide the communication occurring in steps C, D, and E. Step F is the portion of the subdomain boundary calculation that can't be hidden. Typically, the final boundary operation (step F) involves some random memory writes, and it can therefore never be optimized as well as the internal bulk calculations (step B). For smaller subdomain sizes (32^3 per subdomain and less), steps A and F can take longer than step B (the actual bulk calculation).

Results

Now that we've outlined the implementation details, let's consider the results.

Scaling

Figures 2 and 3 show the speedup (versus the same number of CPU cores) and millions of cell updates

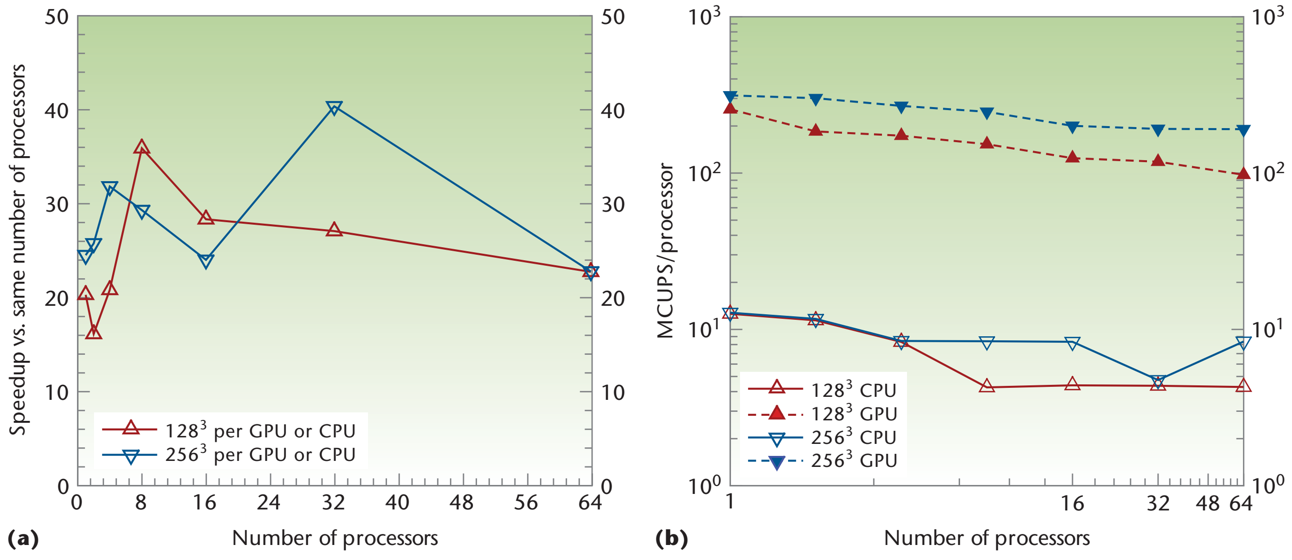

Figure 3. Weak scaling results. (a) Speedup and (b) performance per processor for weak scaling of the 128^3 and 256^3 CFD problem on the Lincoln supercomputer using GPUs and CPUs.

per second per GPU or CPU core (**MCUPS** per processor) for strong and weak scaling results, respectively. MCUPS represents how many millions of finite-volume cells can be updated (one per PPCG iteration) during a second of wall-clock time. In the strong scaling situation, the problem size is constant, so as the number of processors increases, the problem size per processor grows increasingly smaller. For strong scaling, the highest speedup (45 times) occurred for 16 GPUs compared to 16 cores (on four CPUs). In the weak scaling situation, the problem size per processor is constant, so as the number of processors increases, the communication times can grow larger. In theory, the MCUPS per processor is directly related to the hardware efficiency, and should look like a horizontal line (a constant). As Figure 3 shows, from 2 GPUs to 64 GPUs the performance loss is less than 50 percent. (One GPU and one core can each access more memory bandwidth and therefore perform better than when the memory system is loaded to its typical state). Also, using 64 GPUs on the Lincoln supercomputer means using 32 server nodes and therefore four times more network traffic than when computing with 64 CPU cores (which requires using only eight server nodes).

Poisson Solution

Now let's analyze the code's performance on one GPU on Orion. Timings indicate that 87 percent of the code-execution time is spent in the PPCG solver (which solves for the pressure and implicit diffusion terms), and 50 percent of the overall time is spent in the sparse-matrix multiply subroutines alone. Figure 4 gives a breakdown of the time spent on the PPCG and Laplace algorithms. The summation item in Figure 4b includes copying the dot product results from GPU to CPU, plus the last steps of summation on the CPU. As you can see, executing the interior (step B in Figure 1) is the most time-consuming part of the Laplace solver. Copying time plus using the MPI for the largest cases is still four times smaller than interior time. So on Orion, subdomain problem sizes of 128^3 per GPU and larger are sufficient to hide the MPI and copying time. On Lincoln, this isn't actually true, because the copy time is four times slower (the dashed black line in Figure 4b)—so it always is as large as the useful computation time. Current GPUs don't have enough memory to handle problem sizes greater than 288^3 grid points per GPU.

Because every node must send data on its boundaries to other nodes, it must copy six boundary surfaces to the CPU. Thus, steps A, C, D, and E scale the same as N^2. But step B (solving the interior points) grows at the same rate as N^3. As we mentioned, Lincoln has four times slower bandwidth between the CPU and GPU. Figure 4 shows the extrapolated time for copying between the CPU and GPU on Lincoln without MPI time. As you can see, even 256^3 copying times barely overlap with domain computation.

In addition, we used the Nvidia Parallel Nsight tool to analyze the Laplace matrix multiply routine when applied on a Tesla C2070 GPU on Orion. Figure 5 shows the timeline for the Laplace

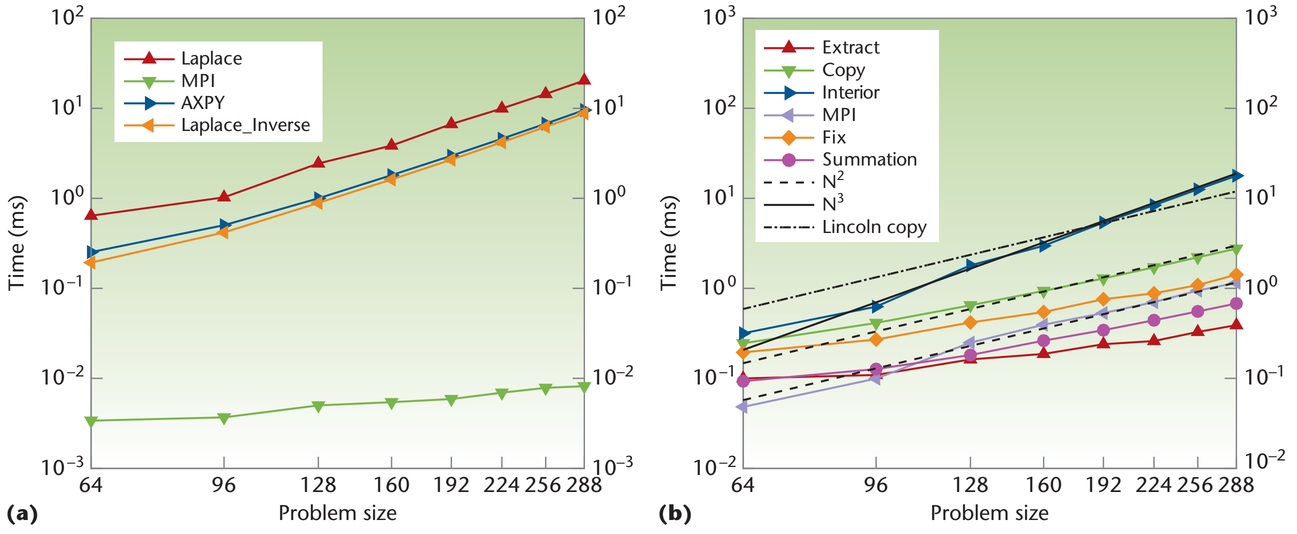

Figure 4. Code performance. Time for the (a) polynomial preconditioned conjugate gradient (PPCG) and (b) Laplace subroutines for different problem sizes. AXPY stands for alpha X plus Y; MPI stands for message passing interface; and N is the problem size (x-axis).

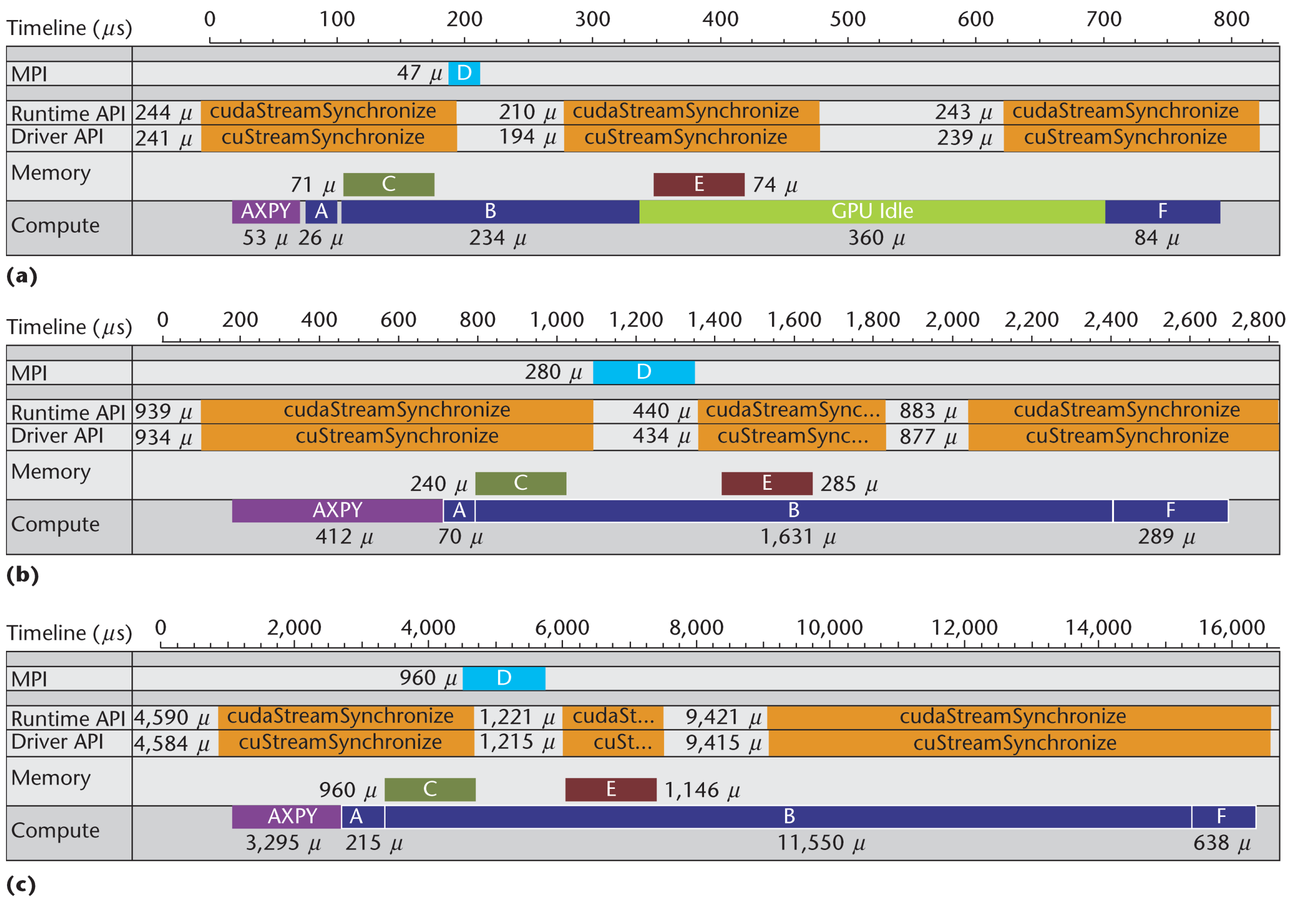

Figure 5. Laplace matrix multiple routine performance on a Tesla C2070 GPU on Orion. Timeline for a Laplace kernel for (a) 64^3, (b) 128^3, and (c) 256^3 problem sizes on one GPU. This figure highlights the fact that for small problem sizes (such as that in Figure 5a) the GPU is so fast at computations (Figure 5b) that the time associated with communication operations (Figures 5c through 5e) can't be hidden.

matrix multiply from this software for problem sizes of 64^3, 128^3, and 256^3.

Figure 5a shows the 64^3 problem size, which doesn't completely hide communication times. Although the actual MPI time overlaps with computation (just barely), the copy back to the GPU doesn't overlap well. In addition, it takes a long time for the boundary fix (step F) to load to the GPU and start executing (the bright green is idle GPU time). This case is strongly affected by the GPU synchronization delays. Synchronization takes on the order of 200 μs to execute, which is the same order of magnitude as the copy commands themselves. In addition, it appears to prevent loading of the `Fix` kernel code to the GPU, which can take yet another 200 μs if it's not performed preemptively.

The latter two simulations show better communication overlapping. On Orion, the 128^3 case even has some leeway for expanded MPI communication times. However, on Lincoln, the two copy times are four times larger and exceed the computation time. Lincoln's successor, called *Forge*, is expected to perform the same as Orion with full PCI Express bandwidth to the CPU memory. For the 128^3 case, the boundary computing time (steps A and F) is 23 percent of the bulk time (reducing the code's effective performance by roughly the same amount). In an ideal calculation, this boundary processing would be as fast as the bulk, and therefore take negligible time (4.7 percent of the bulk).

The 256^3 case shows relatively small communication times compared to the computations. On Lincoln, however, the communication is 75 percent of the computation, and the overlap barely occurs. The boundary-processing time is now down to roughly 8 percent of the bulk computation time. This large problem size involves 67 million unknowns and requires 2.7 Gbytes of memory for double-precision calculations. The Tesla S1070 has 4 Gbytes of memory, so this is roughly the largest problem size possible. Less-expensive retail GPUs typically have only 1.5 Gbytes of memory and a maximum problem size that's about half this simulation size.

We discovered that synchronization can profoundly effect GPU performance. This is particularly true for smaller subdomain sizes, such as 64^3 per GPU. With only one fourth of 10^6 operations, the GPU executes code faster than it can synchronize. The `cudaThreadSynchronize` command sometimes takes 200 ms and can stop the CPU from loading the next executable kernel to the GPU. The `cudaStreamSynchronize` command is similar, and makes the CPU and GPU stall if the specified stream is still running. However, if the stream has already finished, `cudaStreamSynchronize` doesn't take as much time. This code uses implicit synchronization (no explicit commands) as much as possible.

The PCI Express bandwidth is a bottleneck in the multi-GPU performance. Even though the amount of data being sent is small (on the order of 5 percent of the total data being processed by the GPU), the slow speeds make this unusual task a performance bottleneck. From 2007 to 2011, the GPU main-memory bandwidth almost doubled, but the PCI Express speed available on the GPUs hasn't changed.

These bottlenecks leave the GPU with a fairly narrow operating range in terms of the number of unknowns per subdomain that the GPU should process. With less than one million mesh points, CFD calculations don't have enough internal work to hide communication times. And with more than four million mesh points, standard GPUs run out of memory, while Tesla GPUs can go up to four times larger (16 million mesh points). Chunk sizes are also fairly constrained. The number of chunks should be a multiple (two or greater) of the number of multiprocessors. Chunk sizes can vary from $16^2 \times NZ$ to $32^2 \times NZ$, but are probably best sized at $32^2 \times NZ$ to optimize shared (cache) memory use.

Although these observations aren't surprising, they do highlight an emerging problem in scientific computing: common operations are now so fast that they no longer control code performance and unusual operations (related to either chunk or subdomain boundaries) are now so out of balance with these fast operations that they control the performance on everything but the largest problem sizes. We've highlighted this problem here with GPU hardware, but it will soon be a primary issue with CPU hardware as well. In essence, a form of Amdahl's law applies here, which states that small amounts of poorly performing operations can still have a large overall effect on the code's total performance. It's therefore insufficient to design hardware that only speeds up common operations.

Acknowledgments

This work used the US National Science Foundation's (NSF's) Teragrid/XSede supercomputer, Lincoln. The US

Department of Defense primarily supported this project via a subcontract from the Oak Ridge National Laboratory, and we received support from the NSF.

References

1. J.B. Perot and J. Gadebusch, "A Stress Transport Equation Model for Simulating Turbulence at Any Mesh Resolution," *Theoretical and Computational Fluid Dynamics*, vol. 23, no. 4, 2009, pp. 271–286.
2. J.B. Perot, "An Analysis of the Fractional Step Method," *J. Computational Physics*, vol. 108, no. 1, 1993, pp. 183–199.
3. W. Chang, F. Giraldo, and J.B. Perot, "Analysis of an Exact Fractional Step Method," *J. Computational Physics*, vol. 180, no. 1, 2002, pp. 183–189.
4. J.B. Perot and V. Subramanian, "Discrete Calculus Methods for Diffusion," *J. Computational Physics*, vol. 224, no. 1, 2007, pp. 59–81.
5. J.B. Perot, "Conservation Properties of Unstructured Staggered Mesh Schemes," *J. Computational Physics,* vol. 159, no. 1, 2000, pp. 58–89.
6. J.B. Perot, "Discrete Conservation Properties of Unstructured Mesh Schemes," *Ann. Rev. Fluid Mechanics*, vol. 43, 2011, pp. 299–318.
7. J.B. Perot, "Determination of the Decay Exponent in Mechanically Stirred Isotropic Turbulence," *AIP Advances*, vol. 1, no. 2, 2011; doi:10.1063/1.3582815.
8. P. Micikevicius, "3D Finite Difference Computation on GPUs Using CUDA," *Proc. 2nd Workshop General Purpose Processing on Graphics Processing Units*, ACM, 2009, pp. 79–84.

Ali Khajeh-Saeed *is a software engineer at CD-adapco. His research interests include computational fluid dynamics and high-performance computing using general-purpose computation on graphics processing units. Khajeh-Saeed has a PhD in mechanical engineering (with a minor in computer science) from the University of Massachusetts, Amherst. Contact him at khajehsaeed@ecs.umass.edu.*

J. Blair Perot *is the director of the Theoretical and Computational Fluid Dynamics Laboratory at the University of Massachusetts, Amherst. His research focuses on computer simulations of fluid flow and the study of fluid turbulence. Perot has a PhD in mechanical engineering from Stanford University. Contact him at perot@ecs.umass.edu.*