COVER SHEET

NOTE: This conversheet is intended for you to list your article title and author(s) name only—this page will not appear on the CD-ROM.

Title: A Fast Double Precision CFD Code using CUDA

Authors: Jonathan M. Cohen, M. Jeroen Molemaker

PAPER DEADLINE: ***SEPTEMBER 30, 2009****

PAPER LENGTH: ****16 PAGES (Maximum)****

ABSTRACT

We describe a second order double precision finite volume Boussinesq code designed to run on the CUDA architecture. We perform detailed validation of the code on a variety of Rayleigh-Bénard convection problems and show second order convergence. We obtain matching results with a Fortran code running on an eight-core CPU. The CUDA-accelerated code performs approximately eight times faster than the Fortran code on identical problems. As a result, we are able to run a simulation with a grid of size $384^2 \times 192$ at 1.6 seconds per time step on a machine with a single GPU.

INTRODUCTION

Because GPUs are designed for total computational throughput rather than fast execution of serial calculations, they offer the potential for dramatic speedups over multicore CPUs for scientific computing applications. To achieve high computational throughput, GPUs have hundreds of lightweight cores and execute tens of thousands of threads simultaneously, switching between them to hide latencies to off-chip memory. To feed this large number of computational units, GPUs have much higher off-chip bandwidth than do CPUs. For these reasons, they often can achieve high performance for data intensive problems.

While GPUs have a large theoretical performance advantage over CPUs for many problems of interest to the CFD community, there are a number of barriers to their adoption in real-world codes. Modern real-world CFD codes are highly complex software systems representing man-decades or more of development. Because CFD codes require such a large investment, they are designed to last for several hardware generations. Therefore, the potential upside from porting to a new architecture must be large enough to justify the cost of development, and must be expected to persist over multiple hardware generations.

In the short term, many complex numerical codes may adopt an "accelerator" design, where GPUs are used as "bolt-on" units to accelerate performance of

Jonathan M. Cohen, NVIDIA Corporation, Santa Clara, CA 95050, U.S.A.
(e-mail: jocohen@nvidia.com)
M. Jeroen Molemaker, IGPP UCLA, Los Angeles, CA 90095, U.S.A.
(e-mail: nmolem@atmos.ucla.edu)

key bottlenecks. In this design, the majority of computation happens on the CPU. For certain expensive operations, data is copied across the PCI-Express (PCIE) bus to the GPU, where the operation is performed. The result is then copied back to the CPU and the rest of the code proceeds as before. This design has the advantages that it requires minimal changes to an existing code, is straightforward to verify, and allows for an easy fall back path in the case that GPU hardware is not available. However, it suffers from the major problem that data transfers across the PCIE bus are relatively slow, offsetting much of the advantage of high GPU performance.

The accelerator design also suffers from a lack of scalability. Because GPU throughput is roughly linear in the number of floating point units on the chip, it is increasing proportional to Moore's Law, which implies a doubling of transistor density every 12-18 months. Bus bandwidth, on the other hand, is not growing as fast. Therefore, the cost of data transfers is increasing relative to the speed of computation. As this gap between bus speed and transistor density increases, codes which use the accelerator design will fall behind codes that take full advantage of the GPU

The goal of the present work is to explore the use of manycore GPUs to accelerate a globally second order accurate structured CFD code in double precision. We want to look beyond the accelerator approach and design a code from the ground up to run on a GPU. Our code is implemented using CUDA C and is designed to run on an NVIDIA Tesla C1060 GPU. The Tesla C1060 consists of a single GT200 GPU with 240 cores and 4GB of memory GT200 supports IEEE-compliant double precision math with peak throughput of 87 GFLOPS/sec. Detailed benchmarking of our code shows that it is approximately 8 times faster than a comparable multithreaded code running on an 8-core dual-socket Intel Xeon E5420 at 2.5GHz. See Table III for a summary of relative performance.

RELATED WORK

Before programming models like CUDA and similar platforms from other vendors were introduced, fluid dynamics applications running on GPUs were implemented using graphics APIs such as OpenGL or DirectX by recasting numerical problems in terms of rasterization and shader calculations. Despite the difficulty of implementating complex algorithms via graphics APIs, early work in this field such as [9, 2, 10] demonstrated the potential of GPUs for high performance computing.

Since the introduction of CUDA, a number of researchers have demonstrated scientific applications that use GPUs to achieve high performance. In the field of CFD, this includes work on structured grids [20, 25], unstructured grids [4], multiblock codes [5], Euler solvers [3], multilevel methods [6], and discontinuous Galerkin methods [12]. In the related field of atmospheric sciences, [15] demonstrated the use of GPUs to accelerate calculation of a cloud microphysics model in the Weather Research and Forecasting code. While most of this work has focused on the single precision capabilities of GPUs, work by Göddeke *et al.* [8, 7] explored issues with mixed precision solvers for Finite Element calculations on GPUs and other high performance architectures.

NUMERICAL METHOD

We solve the incompressible Navier-Stokes equations using the Boussinesq approximation:

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} &= -(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} - \nabla p + \alpha g T \mathbf{z} \\ \frac{\partial T}{\partial t} &= -(\mathbf{u}\cdot\nabla)T + \kappa\nabla^2 T \\ \nabla\cdot\mathbf{u} &= 0 \end{aligned}$$

where $\mathbf{u} = (u, v, w)$ is the fluid velocity field, T is the fluid temperature, p is the fluid pressure field, α is the coefficient of thermal expansion, g is the magnitude of gravity, ν is kinematic viscosity, and κ is thermal diffusivity.

We solve these equations on a staggered regular grid (Arakawa C-grid) using a second order finite volume discretization. The advection terms are discretized using centered differencing of the flux values, resulting in a discretely conservative second order advection scheme. All other spatial terms are discretized with second order centered differencing.

For time step Δt, we first calculate a new velocity field ignoring the pressure gradient term using a second order Adams-Bashford method:

$$\begin{aligned} \frac{\partial \mathbf{u}_*}{\partial t} &= -(\mathbf{u}\cdot\nabla)\mathbf{u} + \nu\nabla^2\mathbf{u} + \alpha g T \mathbf{z} \\ \mathbf{u}_*^{n+1} &= \mathbf{u}^n + \Delta t \left(\frac{3}{2}\frac{\partial \mathbf{u}_*}{\partial t}^n - \frac{1}{2}\frac{\partial \mathbf{u}_*}{\partial t}^{n-1} \right) \end{aligned}$$

Pressure is treated via a projection method to enforce $\nabla\cdot\mathbf{u} = 0$ at the end of the time step by solving the Poisson equation

$$\Delta t \nabla^2 p = \nabla\cdot\mathbf{u}_*^{n+1} \tag{1}$$

and then subtracting the gradient of p

$$\mathbf{u}^{n+1} = \mathbf{u}_*^{n+1} - \Delta t \nabla p.$$

Equation 1 is solved using a multigrid method [27].

While we have chosen simple discretizations for ease of implementation and validation, our code is designed to support a wide variety of higher-order discretizations and stencils. For wider stencils, the sweep-based technique described in [16] would be more appropriate than the optimizations described in Section 6.6. We note that sweep-based approaches are compatible with the data layout used in our system.

CUDA OVERVIEW

The goal of this work is to demonstrate an implementation of a non-trivial structured CFD code on a GPU. Our implementation uses the CUDA architecture from NVIDIA via the CUDA C front-end. Other front ends such as CUDA Fortran [24] and other GPU programming models such as OpenCL [11] are similar.

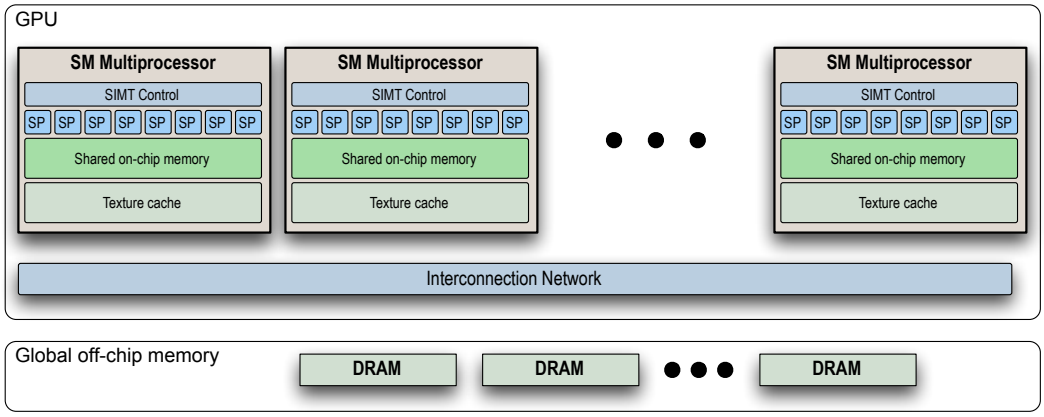


Figure 1. Schematic of the GT200 GPU.

The GT200 GPU [14] consists of a hierarchical array of processors and associated memories, as shown in Figure 1. GT200 consists of 30 *Streaming Multiprocessors* (SMs). Each SM contains 8 cores called *Streaming Processors* (SPs), for a total of 240 cores. Threads are grouped into batches of 32 called *warps* that execute in lockstep *Single Instruction Multiple Data* (SIMD) fashion by running across the 8 cores of an SM at the same time. In this configuration, a single instruction can be executed across 32 threads in 4 clock cycles. Threads within a warp are free to execute any code path. However, because the 8 SPs within an SM share the same instruction fetch, issue, and decode logic, within a 4-clock cycle period only a single instruction can be issued to the 32 threads in a warp. When threads within a warp follow different code paths, there is a performance penalty equal to the number of unique code paths taken. This execution model is referred to as *Single Instruction Multiple Thread* (SIMT).

The Tesla C1060 card consists of a GT200 chip paired with 4GB of global off-chip high-speed DRAM. All SMs can access any value stored in this global memory via load and store instructions. The global off-chip memory, also called *device memory* because it is on the GPU device, has a separate address space from the host CPU's memory. Data can be transferred between host and device memory by crossing the PCIE bus, which typically has lower bandwidth than either the CPU-to-host memory or GPU-to-device memory interfaces.

On the Tesla C1060 card, the GPU-to-device memory bandwidth is about 102 GB/sec (bidirectional). However, this peak bandwidth can only be achieved when all lanes of the DRAM bus are utilized. In order to maximize memory bandwidth, it is necessary to pay careful attention to memory access patterns. A warp can be divided into 2 half-warps of 16 threads each. If threads in a half-warp read from the same 64-byte memory region in the same cycle, these reads are batched into a single operation via a process known as *memory coalescing*. Because coalescing operates at half-warp granularity, uncoalesced loads and stores waste $15/16^{ths}$ of available memory bandwidth. Therefore the most important optimization for memory-bound applications is to arrange work so that threads in the same warp will access sequential memory locations at the same time.

In addition to the device-side off-chip memory, GT200 has two small on-chip caches: a read-only L1 cache called the *texture cache*, and a read/write software managed cache called *shared memory*. With thousand of simultaneous active

threads, on-chip caches are beneficial only if threads scheduled to the same SM access the same cache lines at the same time. Therefore optimizing for cache performance is very similar to optimizing for memory coalescing. The texture cache can be thought of as a "bandwidth aggregator" because it is designed to aggregate memory requests over several cycles so that coalescing will be more efficient. For more information on design of the GT200 caches, see [26].

The CUDA architecture may be programmed via a C/C++ front-end called CUDA C that extends C/C++ with data parallel concepts designed to support easy creation and management a large number of threads. A program that runs on the GPU is called a *kernel*. A kernel is executed by threads, which are divided into a 2-level hierarchy. Individual threads are grouped into batches of up to 1024 called *thread blocks*. Threads within the same thread block are guaranteed to run on the same SM at the same time. There are no ordering, locality, or scheduling guarantees between threads in different blocks. The set of all thread blocks is called a *thread grid*.

Only a single grid may be active on the GPU at the same time, and all threads in a grid must execute the same kernel. A grid of threads is launched via an extended function call syntax that specifies the kernel function, its parameters, the number of threads in each block (which can be specified as a 3D range), and the number of blocks in the grid (which can be specified as a 2D range). The index of a thread within its block, the index of a block within the grid, and the dimensions of the block can all be read by each thread via built-in registers. For more details on the CUDA programming model, see the CUDA programming guide [19].

IMPLEMENTATION

We have explored a number of GPU-specific optimizations for structured CFD codes. We evaluated these optimizations in terms of performance, ease of programming and maintenance, scalability to future hardware, and scalability across difference sizes of data sets. Our goal is to balance all of these factors in order to implement a maintainable but high performance code. Based on our analysis, we recommend the following design strategies:

- remove serial bottlenecks,
- avoid unnecessary PCI-Express transfers,
- run small problems on the CPU,
- layout arrays for maximum memory throughput,
- use congruent padding, and
- use on-chip caches appropriately.

These strategies are described below.

Remove Serial Bottlenecks

We can categorize routines in our code into two types: those whose costs scale with the size of a dataset (*computational routines*), and those whose costs

are fixed regardless of the size of the dataset (*control routines*). In order to make our code scalable to future hardware and to larger data sets, all computational routines are written to take advantage of the parallel GPU architecture.

As the sizes of data sets grow, the fixed cost of control routines will become asymptotically small. Therefore, there is little to gain by parallelizing these routines. On the other hand, any serial implementation of computational routines will come to dominate the total running time. This will limit total possible speedup from improved performance of other computational routines running on parallel GPUs as predicted by Amdahl's Law.

Because of this scaling argument, we want to implement all computational routines to run on the GPU, and all control routines to run on either the CPU or GPU as is convenient. This will give us a maximally scalable code, both to larger data sets and to future GPU architectures with more execution units.

Avoid PCI-Express Transfers

A particular serial bottleneck that we wish to avoid is data transfers across the PCIE bus (PCIE x16 Gen 2 runs at a peak bandwidth of 8 GB/sec). Furthermore, even small data transfers suffer from latency on the order of a few microseconds, regardless of the size of the data payload. While computation can be overlapped with data transfer using CUDA's asynchronous API, this is little help for an iterative solver such as ours running on a single node. Since we cannot easily hide the cost of PCIE transfers, we instead try to remove them entirely. We can achieve this by copying all data to the GPU at the beginning of calculation, and leaving it there for the entire run of the algorithm. We can pull data off for writing to disk using the asynchronous transfer API, which will not slow down computations if properly overlapped.

The consequence of this design decision is that all routines which process simulation data must be implemented to run on the GPU. This includes certain control routines, which may actually run faster on the CPU. In some cases, the performance advantage of the CPU may be high enough to justify the cost of a data transfer, but this has to be evaluated on a case-by-case basis. Our general principal is "first implement on the GPU, then optimize if needed." This is in direct contrast to an accelerator design, where the default is to implement all routines on the CPU and only port computationally expensive routines to the GPU.

Run Small Problems on CPU

CPUs exhibit a non-linear performance curve based on the size of a data set. When the data set required for computation fits entirely in the on-chip caches, CPU performance can be very high. GPUs, on the other hand, are designed to run large numbers of threads and process large datasets. When given a small amount of work with an insufficient amount of parallelism, they can be relatively inefficient. We can exploit these relative performance characteristics by running computational routines on the CPU when it will beat the GPU's performance.

One place in our code where this is relevant is relaxation of the coarse grids in our multigrid solver. As shown in Figure 2, the CPU version of the relaxation routine is 15 times faster on a 4^3 array. For small grids, the time the GPU spends performing actual calculation is small enough that the total elapsed time is dominated by overhead. Therefore, the GPU time stays roughly constant until

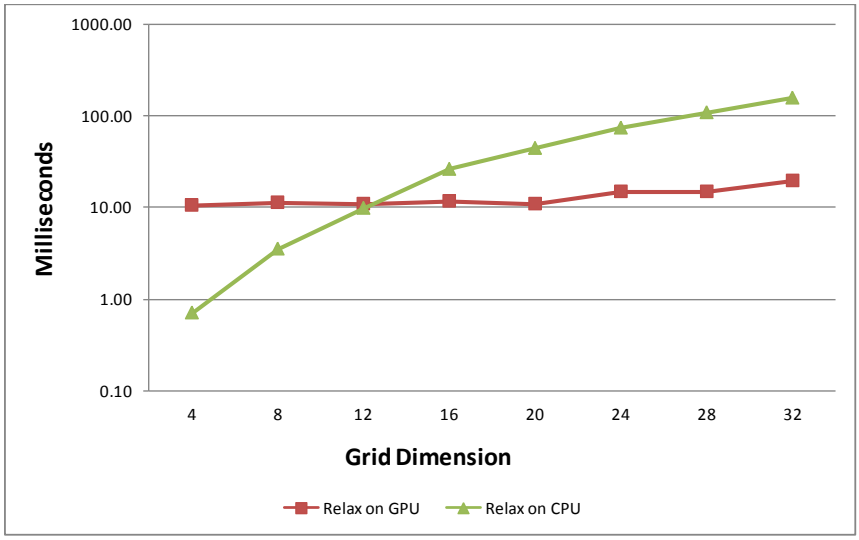


Figure 2. Elapsed time for running 100 relaxation steps on the CPU and the GPU for different grid sizes. Cubic grids were used, so the horizontal labels indicate the lengths of all 3 dimensions.

the 24^3 grid, while the CPU cost grows linearly with the number of total grid cells, crossing over at around 12^3. While the cost of a single relaxation step on the coarse grid is negligible, we perform a large number of coarse grid relaxations in order to solve the Poisson system exactly at this scale. Running the coarsest grid on the CPU results in a 5% reduction in total running time for the $256^2 \times 128$ Rayleigh-Bénard convection problem described in Section 8.

Array Layout

GT200 is designed to run tens of thousands of threads simultaneously, using the large amount of parallelism to hide latencies for off-chip memory reads and writes. The Tesla C1060 has a peak bandwidth to device memory of approximately 102 GB/sec, which corresponds to an optimal balance of just over 6 math operations per double precision value loaded from memory. Because most of our kernels perform a small amount of math, performance of our code is mainly limited by memory bandwidth. Therefore we focused on optimizing memory access patterns to take advantage of GT200's streaming memory architecture.

Most data is stored in 3D arrays. Therefore, it is important to choose a layout for arrays so that the memory system will perform optimally under typical access patterns. We consider small finite difference stencils to be the "typical" case and optimize for them.

To perform array-wide calculation in a data parallel way, each thread updates a single position in the destination array. We can think of every thread as having a 3D index, equal to the index of the element in the array that it will update. Say we choose a thread block size of (b_x, b_y, b_z). To update a 3D array of dimensions (n_x, n_y, n_z), we need to launch a thread grid of $(\lceil n_x/b_x \rceil, \lceil (n_y/b_y)\,(n_z/b_z) \rceil)$ blocks. Note that thread grids are two dimensional, while threads blocks are three dimensional. To overcome this limitation, the y and z dimensions of the grid are folded together and unfolded by each thread using integer modulo and divide.

We made the somewhat arbitrary choice for z to be the fastest changing axis, then y, then x. Because CUDA groups threads into warps based on adjacency in x then y then z, we must transpose between our conceptual thread indices and the hardware notion of thread and block indices. Therefore, in the discussion below we will refer to sequential threads in z as adjacent, even though the hardware considers this dimension to be x.

For simplicity of handling boundary conditions, we pad arrays with (g_x, g_y, g_z) ghost cells in the x, y, and z directions, respectively. This lets us write stencils that access out-of-bounds values at the border of the array as long as g_x, g_y, and g_z are are greater than or equal to the stencil radius. We enforce boundary conditions by filling in these out-of-bounds ghost cells via a separate kernel invocation, which cleanly separates boundary condition handling from the stencil computations. This mechanism could also be used to cleanly insert inter-GPU communication via exchange of ghost values, although we have not yet implemented this feature.

3D arrays are laid out in memory as shown in Figure 3. We choose a simple mapping from 3D array index to memory location that will achieve maximum coalescing when multiple threads access their corresponding array elements at the same time. We pad the beginning of the array so that cell $(-g_x, -g_y, 0)$ is aligned to start at a 64-byte boundary. We pad each z row so that the distance between successive cells in the y direction (which we refer to a *ystride*) is a multiple of 64 bytes. The distance between successive cells in the z direction (referred to as *xstride*) is unconstrained. Therefore, cell $(i, j, 0)$ will always start at a 64-byte boundary, for all i and j. Given the memory location of cell $(0, 0, 0)$, which we call *basepointer*, the location of cell (i, j, k) is

$$basepointer + i * xstride + j * ystride + k. \tag{2}$$

Because *basepointer* starts at a 64-byte boundary, mapping subsequent threads to subsequent z values will result in maximum coalescing when all threads read to or write from their matching array locations.

As an example, consider the kernel that calculates thermal diffusion $\kappa\nabla^2 T$, and adds it to the $\frac{\partial T}{\partial t}$ term, the source code of which is given in Figure 4. Because we use a second order centered discretization, thread (i, j, k) reads 7 values, $T_{i\pm+1,j,k}$, $T_{i,j\pm1,k}$, $T_{i,j,k\pm1}$, and $T_{i,j,k}$. It calculates a Laplacian from these values, then adds the result to the $\frac{\partial T}{\partial t}$ array at position (i, j, k). All reads and writes will be perfectly coalesced, with the exception of the reads from $T_{i,j,k\pm1}$, which will run at half-speed because they are misaligned. In addition to misaligned reads, adjacent threads read overlapping data, which is wasteful of off-chip bandwidth. Using on-chip caches as described below can mitigate both of these effects.

Congruent Padding

We say that two arrays are *congruent* if for all indices (i, j, k), the offset in bytes between the memory location of element $(0, 0, 0)$ and element (i, j, k) is the same for both arrays. Given two arrays, we can pad them with extra elements in x, y, and z to enforce congruency. We refer to this as *congruent padding*. Figure 5 demonstrates congruent padding in two dimensions.

Threads translate from array indices to memory locations using Equation 2. Because they run simultaneously, all threads must calculate all index translations for themselves. Because each thread performs a fairly small amount of

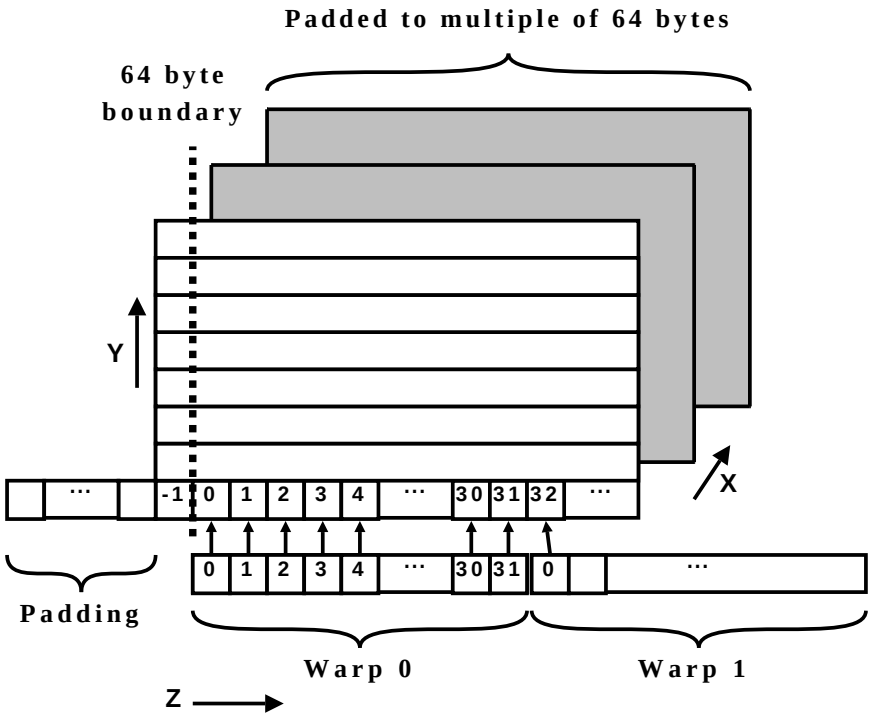


Figure 3. Memory layout of a 3D array, showing how CUDA threads and warps are mapped to array elements. In this example, there is a single row of ghost cell padding on all sides.

TABLE I. Congruent Padding Results

Kernel	Original		With Congruent Padding	
	Registers	Instructions	Registers	Instructions
$-(\mathbf{u}\cdot\nabla)T$	16	132	13	116
$-(\mathbf{u}\cdot\nabla)\mathbf{u}$	46	302	43	264
Multigrid Restriction	14	75	13	68
Multigrid Relaxation	20	327	20	323

work, the cost of this index translation can be high relative to the amount of actual computation performed by a thread. When a single thread accesses several arrays, congruent padding amortizes the cost of index translation by calculating the offset from location $(0, 0, 0)$ to (i, j, k) once per thread, and then adding this offset to the base pointer for each array. In the source code shown in Figure 4, the T and $\frac{\partial T}{\partial t}$ arrays are already congruent allowing for a single index calculation. However, this is not the case for the u, v, and w arrays because the velocity is stored on a staggered grid.

In our system, all arrays are padded to be congruent, regardless of their actual dimensions or the thickness of their ghost cell regions. Table I shows resulting decrease in both register count and number of instructions for several kernels. Reported register usage is the total number of registers used over the entire kernel, and doesn't take into account the reduction in the amount of live registers at any one time. Future GPU architectures may be able to exploit a

```
__global__ void AddThermalDiffusion(
  double *dTdt_baseptr, const double *T_baseptr,
  int xstride, int ystride,
  double invhx2, double invhy2, double invhz2,
  double kappa,
  int blocksInY)
{
  unsigned int blockIdxz = blockIdx.y / blocksInY;
  unsigned int blockIdxy = blockIdx.y % blocksInY;
  unsigned int i = blockIdxz *blockDim.z + threadIdx.z;
  unsigned int j = blockIdxy *blockDim.y + threadIdx.y;
  unsigned int k = blockIdx.x*blockDim.x + threadIdx.x;
  int idx = i*xstride + j*ystride + k;

  double T_ijk = T_baseptr[idx];
  double dTdt_ijk = dTdt_baseptr[idx];
  double laplacianT =
    invhz2 * (T[idx + 1      ] + T[idx - 1      ] - 2.0*T_ijk) +
    invhy2 * (T[idx + ystride] + T[idx - ystride] - 2.0*T_ijk) +
    invhx2 * (T[idx + xstride] + T[idx - xstride] - 2.0*T_ijk);
  dTdt_baseptr[idx] = dTdt_ijk + kappa * laplacianT;
}
```

Figure 4. Source code for the thermal diffusion kernel.

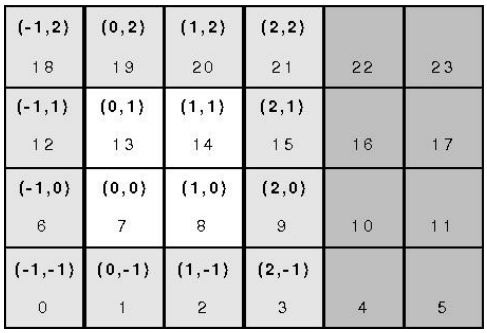
reduction in live register count to improve overall throughput, but this is not currently possible.

On-Chip Caches

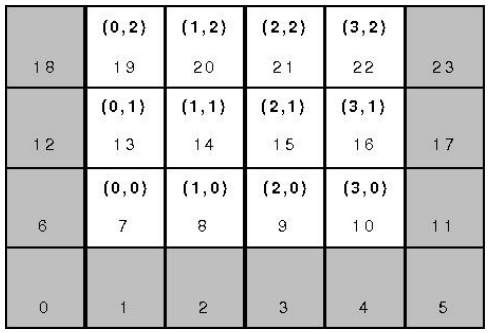
GPU caches serve different purposes from CPU caches. On a multicore CPU, caches perform several functions: capturing data reuse, supporting prefetching to reduce apparent memory latency, and pulling an entire cache line on-chip when the first byte in that cache line is touched to make efficient use of memory interfaces. On a GPU, each thread typically has a small working set and is active for a short period of time, so there is little data reuse to capture. Because a GPU is inherently latency tolerant, there is little advantage to prefetching. The coalescing hardware attempts to batch memory requests to make more efficient use of memory buses.

In our code, we use caches to improve performance by capturing data reuse between nearby threads rather than for a single thread. For many finite difference stencils, there is considerable overlap between the arrays values read by adjacent threads. We can use either shared memory or the texture cache to aggregate loads from several threads into a smaller number of requests, thus reducing memory traffic.

While either texture or shared memory may be used for this purpose, they have different performance characteristics. Because shared memory is software managed, it is flexible enough to often result in optimal bandwidth reduction with careful coding. However, it therefore requires threads to execute additional logic to manage the cache. See [16] for details on how to use shared memory to optimize large 3D finite difference stencils.



(a) A 2x2 array with 1 row of ghost cells on all sides.



(b) A 4x3 array with no ghost cells.

Figure 5. An example of congruent padding in 2 dimensions. Both arrays have the same physical layout in memory, even though they may have different logical dimensions. Computational cells are white, ghost cells are light gray, and unused padding is dark gray.

TABLE II. Calculated critical Rayleigh values for full-slip (aspect ratio $\sqrt{2} : .5 : 1$) and no-slip (aspect ratio $\pi : .5 : 3.11$) boundaries at different resolutions.

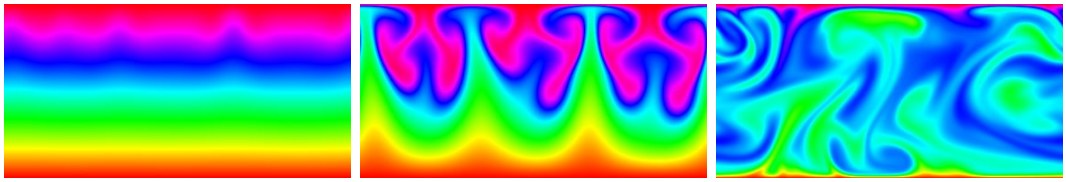
Resolution	Full Slip		No Slip	
	Value	Diff	Value	Diff
$16 \times 8 \times 16$	659.69	–	1674.29	
$32 \times 16 \times 32$	658.05	1.64	1699.25	24.96
$64 \times 32 \times 64$	657.65	0.40	1705.59	6.34
$128 \times 64 \times 128$	657.54	0.11	1707.22	1.63
∞	657.51	–	1707.76	–

The texture cache is hardware managed. While it may not reduce bandwidth as much as shared memory, it is easier to use because it does not increase code complexity. Because logic to manage the texture cache is implemented in hardware, it can actually be faster than shared memory in some cases, even when its bandwidth reduction is less.

In general, we have found that using the texture cache improves performance of most finite different kernels by a factor of 1.5. We believe texture cache is a good compromise between code complexity and performance, and therefore use texture cache in almost every routine where there is overlap between adjacent thread reads. These results are similar to those obtained by [1], which computed sparse matrix-vector products with CUDA.

VALIDATION

To demonstrate the potential of GPU based codes for scientific applications we have validated our code on a range of problems. We compared our results with an existing CPU based code written in Fortran [18] as well as with published analytical, numerical, and experimental results. Since our code implements the Boussinesq equations we choose to examine whether it can reproduce known



(a) Perturbations appear in the stratifications.

(b) Instabilities form.

(c) Temperature mixes after the onset of turbulence.

Figure 6. False color plot of T at the $y = 0$ plane for a $384^2 \times 192$ resolution simulation with $Ra = 10^7$.

solutions to different Rayleigh-Bénard convection problems in which a constant temperature difference ΔT is maintained between the top and bottom boundaries of the domain. The most basic result for Rayleigh-Bénard convection is the critical value of the dimensionless Rayleigh number $Ra = g\alpha\Delta T/\kappa\nu$. Below the critical value Ra_c the solution is motionless and heat flux between top and bottom is purely diffusive. When $Ra > Ra_c$ the diffusive solution becomes unstable to perturbations of arbitrarily small amplitude and a solution with non-trivial flow and enhanced vertical heat transport ensues.

We estimated Ra_c in our codes by calculating positive and negative growth rates of $\mathbf{u}$ for small perturbations around Ra_c and extrapolating to find the value of Ra for which the growth rate would be zero. Our GPU and CPU codes use identical numerical methods and therefore have matching values to several decimal places. Table II shows calculated Ra_c values from our codes. The third and fifth columns (labeled Diff) show the differences in Ra_c obtained for subsequent resolutions. The reduction of this error by a factor of 4 for each doubling of resolution shows the globally second order convergence character of the numerical discretizations. For the 2D problem with the aspect ratios chosen, analytical values are known [22] (we treat this as a 3D problem by choosing a smaller aspect ratio and periodic boundaries in the y dimension). Using Richardson extrapolation, we obtain a value of $Ra_c = 657.51$ for the full-slip case, and $Ra_c = 1707.76$ for the no-slip case, both of which match the analytical results.

To test a fully 3D problem, we also studied the onset of convection in a cubic box with no-slip conditions on all sides and Dirichlet conditions for T on the side boundaries. We found a critical Rayleigh number $Ra_c = 6755$ for this case, matching published experimental [13] and numerical [17, 21] values. To verify the nonlinear advection terms in the equations, we calculated the solution for a supercritical value of $Ra = 4.4 \times 10^4$. We then calculated the Nusselt number $Nu = \overline{(wT + \kappa T_z)}/(\kappa\Delta T)$. The Nusselt number is the ratio of vertical heat transport to diffusive heat transport across a 2D interface of a motionless solution. Nu also depends on the Prandtl number, $Pr = \nu/\kappa$. For $Ra = 4.4\times10^4$ and $Pr = 0.71$, Nu computed at the upper and lower boundaries is 2.05 (using both CPU and GPU codes) and exhibits global second order convergence when computed at increasing resolutions. This matches published results [21].

TABLE III. Relative performance of the CPU and GPU codes on the $Ra = 10^7$ problem.

CPU Fortran Code			
Resolution	ms/step	ms/step/node	Scaling
$64^2 \times 32$	47	37.0e-5	-
$128^2 \times 64$	327	31.2e-5	0.84x
$256^2 \times 128$	4070	48.5e-5	1.55x
$384^2 \times 192$	13670	48.3e-5	1.00x

GPU CUDA Code				**GPU Speedup**
Resolution	ms/step	ms/step/node	Scaling	
$64^2 \times 32$	24	18.3e-5	-	**2.0x**
$128^2 \times 64$	79	7.5e-5	0.41x	**5.3x**
$256^2 \times 128$	498	5.9e-5	0.79x	**8.2x**
$384^2 \times 192$	1616	5.7e-5	0.97x	**8.5x**

TABLE IV. Relative performance of double and single precision GPU codes on a lock exchange problem.

Resolution	seconds/step (fp64)	seconds/step (fp32)	Ratio
64^3	0.020912	0.014367	1.46
128^3	0.077741	0.046394	1.68
256^3	0.642961	0.387173	1.66

PERFORMANCE ANALYSIS

To generate a timing comparison, we ran an unsteady Rayleigh-Bénard convection problem on our GPU and CPU codes with $Ra = 10^7$ and $Pr = .71$. The simulation domain was set to $[-1, 1] \times [-1, 1] \times [-.5.5]$, with periodic boundary conditions in x and y, and no-slip boundaries in z. The pressure solver was run until divergence was below 10^{-8}.

As shown in Figure 6, the flow starts out motionless until instabilities form, and then transitions to turbulence. To accelerate convergence of the multigrid solver for pressure, we reuse the solution from the previous time step as an initial guess. Consequently, the number of v-cycles required for convergence increases as the flow becomes less steady. In order to characterize our performance fairly, we only count the average time per step once the number of v-cycles per step has stabilized.

Because of the different performance characteristics of the GPU and the CPU, we have chosen different multigrid relaxations schemes for the two codes. The GPU code, which uses a red-black Gauss-Seidel point relaxer, requires 1 full-multigrid step followed by 7 v-cycles at all resolutions. The CPU code uses a red-black line relaxer and requires 1 full-multigrid step followed by 13 v-cycles. Table III shows the relative timing of the two codes at different resolutions. GPU times are for a single Tesla C1060 running on a Core2-Duo E8500 at 3.17GHz, CPU times are for an 8-core dual-socket Xeon E5420 at 2.5GHz.

Figure V shows the ten most expensive routines of the CUDA code running

TABLE V. Ten most expensive routines in CUDA code

24.97%	MultigridPressure3DDeviceD_relax(256)
8.69%	LaplacianCentered3DDevice_stencil
8.02%	Advection3DD_apply_stencil
7.36%	MultigridPressure3DDeviceD_calculate_residual(256)
6.76%	MultigridPressure3DDeviceD_relax(128)
5.70%	MultigridPressure3DDeviceD_prolong(128)
5.51%	Grid3DDevice_linear_combination
3.35%	ThermalAdvection3D_apply_stencil
3.19%	MultigridPressure3DDeviceD_apply_boundary_conditions(256)
2.76%	Grid3DDevice_reduction

the $256^2 \times 128$ problem. It is well balanced, with the most expensive routine, multigrid relaxation, accounting for the largest percentage of time (approximately 35% including all grid levels and boundary condition updates). PCIE transfers do not even appear in ten most expensive routines, indicating that we have effectively removed them as a bottleneck. The profiling results justify our design decision to use texture cache for many of the finite difference stencil routines even though it is less optimal than shared memory. For example, while the `LaplacianCentered3DDevice_stencil` routine could be optimized with careful use of shared memory, the overall code speedup could be no more than 8.69%, and would likely be much smaller. Because texture cache achieves reasonable performance with almost no extra code complexity, we use it extensively.

We are also interested in understanding the suitability of the GT200 for double precision calculations. In terms of theoretical peak performance, GT200 has 12 times more throughput in single precision (936 GFLOPS) than double precision (78 GFLOPS). This is due to a smaller number of double precision units (only 1 per SM, rather than 8 per SM), as well as a lower issue rate [19]. However, differences in theoretical peak performance do not tell the whole story, and we are interested in actual numbers for a complete application.

To compare the relative performance of using double precision versus single precision on GT200, we ran a lock exchange problem as a simple benchmark. We use a unit cube box with no-slip boundaries on all sides, and set the initial conditions to $\mathbf{u} = 0$ and $T = 2 \tanh x$, with $\kappa = 1$ and $\nu = 1$. In order to make a fair comparison, we chose a convergence tolerance of 10^{-4} for the multigrid pressure solver, since this is achievable with single precision arithmetic. We ran several time steps using both single precision (fp32) and double precision (fp64) versions of the code, and calculated the average seconds per time step, as reported in Table IV. Across the different resolutions, the double precision version is only 46% to 66% slower than the single precision version, rather than 12 times slower.

CONCLUSIONS

Our results demonstrate that GPU-based codes can be powerful tools for real scientific applications. We have demonstrated second order convergence in double precision on buoyancy-driven turbulence problems using a GPU. Using a single workstation that is equipped with a GPU, our code can integrate a

non-trivial flow at moderate resolutions up to 8 times faster than an 8-core CPU. It is interesting to note that the speedup we see, about a factor if 8, is similar to the speedups observed by Bell and Garland for sparse matrix-vector multiplication [1].

We posit that this speedup is largely a result of the high bandwidth from the GPU processors to the GPU memory. The GPU-to-device memory bandwidth is approximately an order of magnitude higher than CPU-to-host memory bandwidth. For large data sets, CPU caches do not reduce latencies because the working set is too large to fit in even L3 cache. Because the numerical methods we use have fairly low arithmetic intensity, performance is largely limited by bandwidth between processors and memory, which explains the performance results we observe.

The difference between double precision and single precision tells a similar story. Because we are limited by memory bandwidth, the floating point units are not saturated, especially in single precision. Therefore, the performance difference is largely explained by the fact that double precision requires twice as much memory. This indicates that for bandwidth-limited applications, GT200 can be expected to perform well on double precision problems, despite the large disparity in peak theoretical throughput of the floating point units.

Taking advantage of increased memory bandwidth and processing width requires careful implementation to run at peak memory efficiency, and to avoid serial bottlenecks which will come to dominate running time. An "accelerator" design does not remove serial bottlenecks, and does not allow changes to data layout which are necessary for taking advantage of the GPU memory system. For these reasons, it would be difficult to obtain such high performance numbers if we merely ported the bottlenecks of the Fortran code to CUDA. Rather, we have demonstrated that a ground-up rewrite is possible, and leads to a design that can achieve very high performance.

We intend to extend our work in several ways. First, we will implement higher-order methods in both space and time. Second, we are interested in numerical ocean and atmospheric models such as [23] that use logically regular grids, but are geometrically irregular. Third, we will explore multiple GPU configurations such as [25]. A single motherboard may have several PCIE buses, each of which can connect to one or more GPUs. In addition to improving performance for large computing clusters, this has the potential to dramatically increase the resolution that people without access to clusters can achieve.

REFERENCES

1. N. Bell and M. Garland. 2009."Implementing sparse matrix-vector multiplication on throughput-oriented processors," *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing* New York, NY, USA: ACM.
2. J. Bolz, I. Farmer, E. Grinspun, and P. Schroder. 2003. "Sparse matrix solver on the GPU: Conjugate gradients and multigrid," *Proceedings of ACM SIGGRAPH 2003, ACM Transaction on Graphics*, vol. 22, pp. 917–924.
3. T. Brandvik and G. Pullan. 2009. "Acceleration of a 3d Euler solver using commodity graphics hardware," 46^{th} *AIAA Aerospace Sciences Meeting*, paper no: AIAA-2008-607.
4. A. Corrigan, F. Camelli, R. Löhner, and J. Wallin. 2009. "Running unstructured grid CFD solvers on modern graphics hardware," *19th AIAA Computational Fluid Dynamics Conference*, no. AIAA 2009-4001.
5. E. Elsen, P. LeGresley, and E. Darve. 2008. "Large calculation of the flow over a hypersonic vehicle using a GPU," *J. Comput. Phys.* **227** (2008), no. 24, 10148–10161.

6. D. Göddeke, S. H.M. Buijssen, H. Wobker, and S. Turek. 2009. "GPU acceleration of an unmodified parallel finite element Navier-Stokes solver," *High Performance Computing & Simulation 2009* (Waleed W. Smari and John P. McIntire, eds.), pp. 12–21.
7. D. Göddeke and R. Strzodka. 2008. "Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (part 2: Double precision GPUs)," Tech. report, Technical University Dortmund.
8. D. Göddeke, R. Strzodka, and S. Turek. 2007. "Performance and accuracy of hardware-oriented native-, emulated-and mixed-precision solvers in fem simulations," *Int. J. Parallel Emerg. Distrib. Syst.* **22** (2007), no. 4, 221–256.
9. N. Goodnight, C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. 2003. "A multigrid solver for boundary value problems using programmable graphics hardware," *Graphics Hardware* (2003), 102–135.
10. T. Runar Hagen, K.-A. Lie, and J. R. Natvig. 2006. "Solving the Euler equations on graphics processing units," *Computational Sciences - ICCS 2006* **3994** (2006), 220–227.
11. Khronos OpenCL Working Group. 2008. *The OpenCL specification*, Version 1.0.
12. A. Klöckner, T. Warburton, J. Bridge, and J. S. Hesthaven. 2009. "Nodal discontinuous Galerkin methods on graphics processors," *J. Comput. Phys.* **228** (2009), no. 21, 7863–7882.
13. W. H. Leong, K. G. T. Hollands, and A. P. Brunger. 1998. "On a physically realizable benchmark problem in internal natural convection," *Int. J. Heat Mass Transfer* **41** (1998), 3817–3828.
14. E. Lindholm, J. Nickolls, S. Olberman, and J. Montrym. 2008. "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro* **28(2)** (2008), 39–55.
15. J. Michalakes and M. Vachharagani. 2008. "GPU acceleration of numerical weather prediction," *Parallel Processing Letters* **18** (2008), no. 4, 531–548.
16. P. Micikevicius. 2009. "3d finite difference computation on GPUs using CUDA," *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA), ACM, 2009, pp. 79–84.
17. J. Mizushima and O. Matsuda. 1997. "Onset of 3d thermal convection in a cubic cavity," *J. Phys. Soc. Japan* **66** (1997), 2237–2341.
18. M. J. Molemaker, J. C. McWilliams, and X. Capet. 2009. "Balanced and unbalanced routes to dissipation in equilibrated Eady flow," *J. Fluid Mech.* (2009), In press.
19. NVIDIA Corporation. 2008. *CUDA programming guide*, Version 2.3.
20. E. H. Phillips, Y.Zhang, R. L. Davis, and J. D. Owens. 2009. "Rapid aerodynamic performance prediction on a cluster of graphics processing units," *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, no. AIAA 2009-565, January 2009.
21. D. Puigjaner, J. Herrero, C. Simo, and F. Giralt. 2008. "Bifurcation analysis of steady Rayleigh-Bénard convection in a cubical cavity with conducting sidewalls," *J. Fluid Mech.* **598** (2008), 393–427.
22. W. H. Reid and D. L. Harris. 2005. "Some further results on the Bénard problem," *Phys. Fluids* **1** (1958), 102–110.
23. A. F. Shchepetkin and J. C. McWilliams. 2005. "The regional oceanic modeling system (ROMS): a split-explicit, free-surface, topography-following-coordinate oceanic model," *Ocean Modelling* **9** (2005), 347–404.
24. The Portland Group. 2009. *CUDA Fortran programming guide and reference*, Version 0.9.
25. J. C. Thibault and I. Senocak. 2009. "CUDA implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows," 47^{th} *AIAA Aerospace Sciences Meeting*, 2009, paper no: AIAA-2009-758.
26. V. Volkov and J. W. Demmel. 2008. "Benchmarking GPUs to tune dense linear algebra," *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing* (Piscataway, NJ, USA), IEEE Press, 2008, pp. 1–11.
27. I. Yavneh. 1996. "On red-black SOR smoothing in multigrid," *SIAM J. Sci. Comput.* **17** (1996), no. 1, 180–192.