J Real-Time Image Proc (2010) 5:179–193
DOI 10.1007/s11554-009-0133-1

ORIGINAL RESEARCH PAPER

Orders-of-magnitude performance increases in GPU-accelerated correlation of images from the International Space Station

Peter J. Lu · Hidekazu Oki · Catherine A. Frey · Gregory E. Chamitoff · Leroy Chiao · Edward M. Fincke · C. Michael Foale · Sandra H. Magnus · William S. McArthur Jr. · Daniel M. Tani · Peggy A. Whitson · Jeffrey N. Williams · William V. Meyer · Ronald J. Sicker · Brion J. Au · Mark Christiansen · Andrew B. Schofield · David A. Weitz

Received: 16 March 2009 / Accepted: 28 September 2009 / Published online: 30 October 2009
© Springer-Verlag 2009

Abstract We implement image correlation, a fundamental component of many real-time imaging and tracking systems, on a graphics processing unit (GPU) using NVIDIA's CUDA platform. We use our code to analyze images of liquid-gas phase separation in a model colloid-polymer system, photographed in the absence of gravity aboard the International Space Station (ISS). Our GPU code is 4,000 times faster than simple MATLAB code performing the same calculation on a central processing unit (CPU), 130 times faster than simple C code, and 30 times faster than optimized C++ code using single-instruction, multiple-data (SIMD) extensions. The speed increases from these parallel algorithms enable us to analyze images downlinked from the ISS in a rapid fashion and send feedback to astronauts on orbit while the experiments are still being run.

Keywords GPU · CUDA · Autocorrelation · International Space Station · SIMD

P. J. Lu (✉) · D. A. Weitz
Department of Physics and SEAS, Harvard University, Cambridge, MA, USA
e-mail: plu@fas.harvard.edu

H. Oki
Shinagawa-ku, Tokyo, Japan

C. A. Frey
ZIN Technologies, Inc., Middleburg Heights, OH, USA

G. E. Chamitoff · L. Chiao · E. M. Fincke · C. M. Foale · S. H. Magnus · W. S. McArthur Jr. · D. M. Tani · P. A. Whitson · J. N. Williams
International Space Station, Low Earth Orbit, and NASA Johnson Space Center, Houston, TX, USA

W. V. Meyer · R. J. Sicker
NASA Glenn Research Center, Cleveland, OH, USA

B. J. Au
United Space Alliance and NASA Johnson Space Center, Houston, TX, USA

M. Christiansen
Flowseeker LLC, San Francisco, CA, USA

A. B. Schofield
The School of Physics and Astronomy, University of Edinburgh, Edinburgh, UK

1 Introduction

Most computer programs are serial, where the results of one stage of computation are used as input for the next. Since they execute on a computer's central processing unit (CPU), their performance is determined by CPU clock speed. For the early years of CPU evolution, clock speeds increased exponentially, but over the past decade this rate increase has not persisted; instead, computers have included multiple CPU cores to increase computational power. Unfortunately, serial code cannot automatically take advantage of multiple cores to execute more quickly, so that software performance has not in general kept pace with hardware in this period.

Nevertheless, some applications commonly used by scientists and engineers can use multiple cores in parallel for specialized operations. MATLAB, for example, uses parallel processes on multiple cores to accelerate fast Fourier transforms (FFTs) and certain aspects of image processing; these calculations are ideal for parallelism because the same operation is applied independently to different pieces of data. Software development tools have also simplified the process of parallelizing code, including specialized libraries [43] and modern compilers that parallelize certain operations automatically, without requiring additional work by the programmer [5]. While convenient and simple to use, these CPU-based parallel approaches are ultimately limited by the number of processor cores, rarely more than four; the resulting performance increases are generally less than an order of magnitude [5].

By contrast, all modern PCs come equipped with a specialized multiprocessor, a graphics processing unit (GPU) containing up to hundreds of simplified processor cores—opening up the tantalizing possibility of far greater speed increases. In fact, although they are designed and heavily optimized for rendering pixels on a screen, GPUs can also be harnessed for general computation in a modality known as general-purpose GPU (GPGPU) [29–31]. In the earliest GPGPU applications, the GPU performed operations on regular data formatted as graphics objects [11, 31], a tricky programming task that limited its use to graphics experts.

In recent years, however, GPGPU computing has been revolutionized by the advent of general programming languages implemented on the GPU [28]; an excellent overview of the different approaches is given in [30]. Of particular present interest are the CUDA C [48] and OpenCL [49] languages supported by the CUDA architecture. CUDA C is a set of parallel extensions to the C/C++ programming languages and interacts with a special hardware interface built into all current NVIDIA GPUs; within CUDA, the GPU appears to the programmer as a collection of general-purpose parallel processors [29, 48]. Specific knowledge of graphics programming is no longer necessary for GPGPU; as a result of CUDA, the number of GPGPU applications in science and engineering has increased significantly [14]. Impressive performance gains have been achieved in several types of CUDA simulations [29], including molecular dynamics [2, 20, 41, 45], Monte–Carlo [1], N-body gravitation [4], Navier–Stokes [44], lattice quantum chromodynamics [17], particle-in-cell plasma [40] and schooling fish [18]. A number of computational algorithms and numerical calculations have also been implemented in CUDA [9], including parallel sorting [39], binary-string search for digital forensics [27], sparse-matrix factorization [36], fast-multipole methods [15] and diffraction integral calculations [37]. CUDA applications that analyze experimental data mostly involve information and image processing [8], including matching DNA sequences [26, 35], correlating cosmological data [32] and radio-astronomy telescope output [16], laser-speckle imaging of blood flow [19], digital holographic microscopy [38], face tracking [21], magnetic resonance image (MRI) reconstruction [42], and aligning and deforming microscopy images for 3D reconstruction [33, 34]. Using CUDA, most of these applications accelerate performance by one to two orders of magnitude over that of a single CPU, far more than usually possible with only four CPU cores [48].

In this paper, we present several parallel implementations of a fundamental image-analysis calculation, spatial autocorrelation. We step through its expression in a simple MATLAB code, a simple C code, and an optimized C++ version that uses single-instruction, multiple-data (SIMD) extensions to improve CPU-based calculation speed. We then present our CUDA C implementation, and demonstrate how the GPU performs the same calculation several orders of magnitude faster. We characterize and explain the strengths and weaknesses of the approaches taken in the different codes, emphasizing underlying concepts for scientists and engineers who may not be expert programmers. We apply our code to analyze images of liquid–gas phase separation, collected in the low-gravity environment by astronauts aboard the International Space Station (ISS) [3]. Our accelerated parallel algorithms enable rapid analysis of images downlinked from ISS to earth, allowing us to provide feedback to the astronauts on orbit in time to make changes while the experiment is still running. This interactive mode of operation is a significant departure from most other ISS experiments, which are typically configured to run in an automated fashion and cannot be modified after launch.

2 Materials and methods

2.1 Sample preparation and photography

Our colloid-polymer sample preparation procedure is described in detail in chapter 12 of [25]. We suspend polymethylmethacrylate spheres in a solvent mixture that nearly matches the particles' index of refraction. The samples are transparent when viewed straight on, but scatter blue light at a high angle. We induce phase separation by adding a linear polystyrene, which creates a depletion attraction between the particles, which we tune chemically to mimic the role of temperature in molecular systems [22–24, 47]. Our samples have polymer and colloid concentrations near where previous experiments have extrapolated the liquid–gas critical point [6, 7].

2.2 Performance testing

Differences in CPU hardware limit useful comparison of CUDA speedups to order-of-magnitude estimates. While GPUs are fairly standard (G80 or GT200), almost every paper compares to a different CPU; in general, Pentium 4 CPUs are roughly twice as slow, and Core i7 CPUs twice as fast, as Core2 CPUs at the same clock speed. For our testing platform, we selected a middle-of-the-road consumer desktop PC configuration likely to represent what is found in a typical laboratory: a Dell XPS 730 with an Intel Q6600 Core2 Quad at 2.4 GHz, with 3GB of RAM and WinXP SP2. We observe the same GPU performance on the Tesla C1060, Quadro FX 5800, and reference-design GeForce GTX 280 graphics cards. Because our code uses an atomic operation, it requires a GPU with CUDA Compute Capability level 1.1 or greater; in particular, the G80 series of cards, with capability level 1.0, cannot run this code in its entirety. An earlier version of our code therefore wrote intermediate values back to global memory, which were summed using a second kernel; this leads to an overall decrease in performance of 25%, relative to the GPU results presented in Fig. 9.

We compare results for total program execution times on a single CPU core and single GPU, motivated by the hardware of recent GPU clusters [32]. Because autocorrelation involves only one image at a time, the most efficient way to parallelize is to designate each CPU core or GPU to separately analyze a different image from the time sequence; OpenMP provides a convenient and straightforward way to implement this parallelization, though the data handling becomes slightly more complicated. We have achieved the expected, nearly fourfold performance improvement for the C++ algorithm using OpenMP and executing the code on the quad-core processor, relative to the single core, non-OpenMP version. However, a nearly identical speedup would be achieved by using OpenMP to send the images to four graphics cards, which could take the form of two GeForce GTX 295 cards; these cards would provide the four GT200 GPUs to match the four CPU cores in the quad-core processor. This perfect scaling applies equally to all codes, so that the relative speedups shown in Fig. 9 will remain unchanged.

3 Photographing phase separation onboard the International Space Station

Understanding the separation of liquid and gas phases is a fundamental problem with myriad practical applications, including the formation of clouds, the decompression of liquid fuel in a rocket engine, and the spray from an aerosol can. The liquid form is denser than the gaseous form of the same material, so investigating the structures formed in this process is difficult in the presence of gravity on earth. We therefore launched a series of liquid–gas mixtures to the ISS as part of the BCAT3 and BCAT4 experiments, where the effects of gravity are reduced by six orders of magnitude. Our samples comprise colloids and polymers diffusing in a background solvent; this model system mimics the behavior of molecular liquids and gases using larger particles that we photograph, and whose interactions we control with chemistry [24, 47].

Samples are loaded into glass cuvettes, which are then placed inside a sample holder, shown in Fig. 1. Inside each cuvette is a tiny teflon stir bar, which is agitated to mix the colloids and polymers. The samples then separate into colloid-rich liquid and colloid-poor gas phases, much like salad dressing separates into an oil layer and a water layer. We photograph this process with a digital SLR camera (Kodak DCS760) and flash (Nikon SB28) controlled remotely via firewire from a laptop running EarthKAM software, which inputs a list of times to trigger the camera, then automatically downloads the resulting images without astronaut intervention. The photography setup onboard the ISS is shown in Fig. 2.

We usually take a photograph every hour for a week, generating a few hundred images of the sample evolution throughout the course of a complete data acquisition run. Each day, we receive a couple dozen images, downlinked from ISS, showing how the sample evolved during the previous day; we analyze these images immediately to monitor the progress of the experiment. Based on these results, we can communicate feedback to the astronauts on ISS, who can then alter the experiment, if necessary, before the next day's data are collected and downlinked. A typical raw, unprocessed image from the middle of one of these sequences is illustrated in Fig. 3a. We use the Adobe Camera RAW converter [12] to import the camera's CCD data stored in the Camera RAW format into Adobe Photoshop CS3, where we convert the image to grayscale, as illustrated in Fig. 3b. To remove the random motion due to drift over time in the relative positions of camera and sample, we stabilize the converted grayscale image sequence in Adobe After Effects CS3 [10], which precisely registers and aligns all images in the time series.

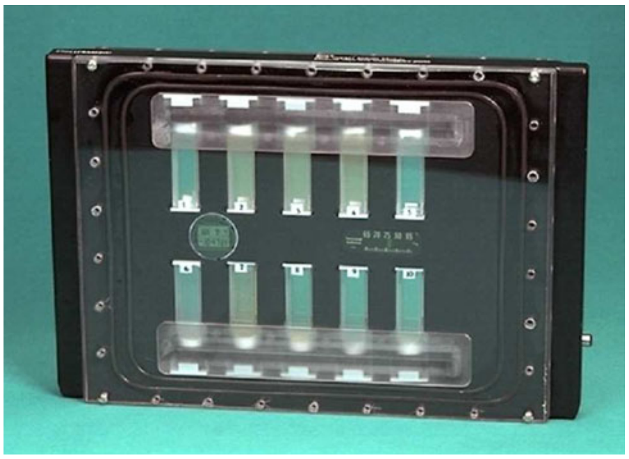
Fig. 1 BCAT sample holder with ten glass cuvettes in an anodized aluminum frame, between two sheets of plexiglass

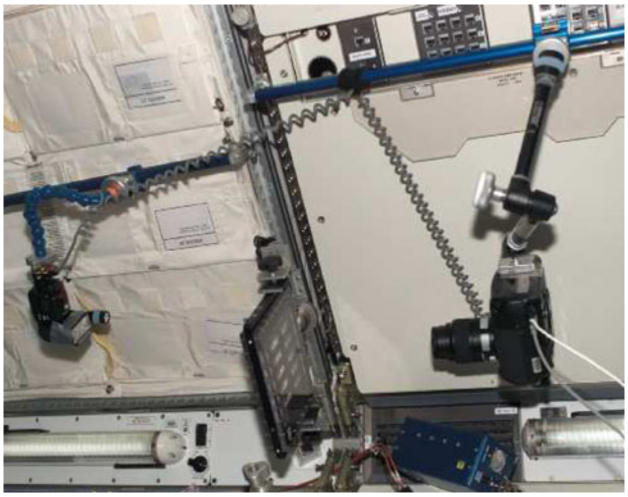
Fig. 2 Camera setup onboard the ISS, clamped to bars that attach to the walls and ceiling for greater stability

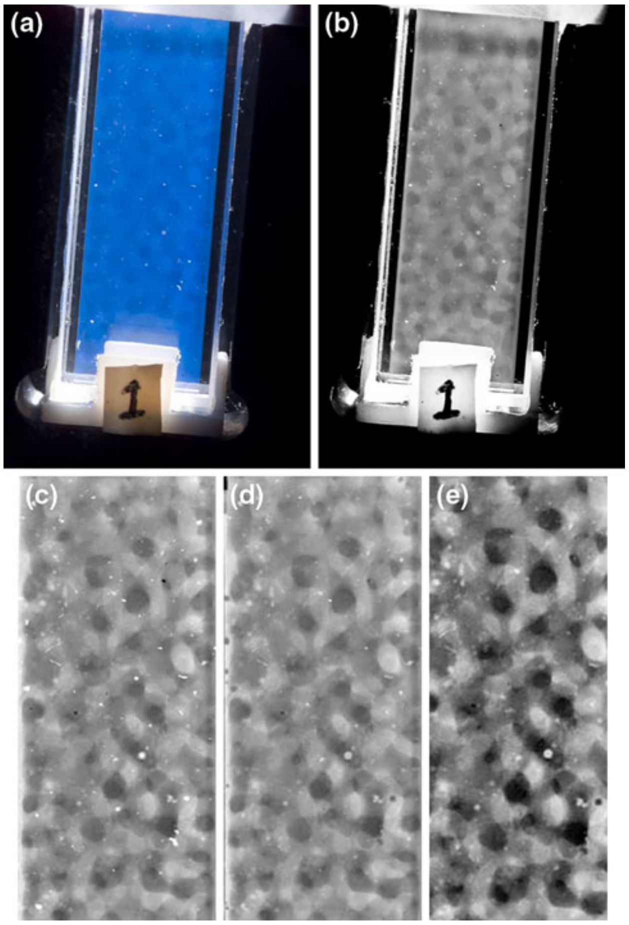

Fig. 3 Automated processing of photographs of liquid-gas phase separation collected onboard the ISS as part of the BCAT3 experiment. The colloid-rich liquid is the brighter phase; the colloid-poor gas appears darker. **a** Original image before processing. **b** Grayscale image after conversion in Camera Raw. **c** Cropped image after uneven lighting background has been removed. **d** Image after automated dust removal. **e** Final image, after further cropping and final contrast enhancement

We remove the uneven lighting background by blurring an inverted copy of the image on a separate layer, using the Overlay blending mode with 50% opacity, resulting in the image in Fig. 3c. To remove the dust on the camera sensor and in the sample, we use the Color Key Filter to select the white and black pixels and replace these regions with copies of the original image offset by a few pixels, maintaining the correct brightness and noise distributions, as shown in Fig. 3d. In the last step, we enhance contrast and crop, yielding the final image shown in Fig. 3e. All intermediate steps are implemented as 16-bit nested (precomposed) projects in After Effects, maximally preserving image information before final output. The resulting images, typically 750 × 1,500 pixels, are converted in Photoshop to losslessly compressed 8-bit grayscale PNG format and form the data input for the autocorrelation calculation.

4 Image autocorrelation analysis

The network structure that appears in the phase separation images has a characteristic length scale, which in Fig. 3 is about one-eighth of the width of the sample. How this length changes with time gives insight into the thermodynamics driving the phase separation [13]. To quantify this length, we calculate the 2D intensity autocorrelation [3]:

$$C_{2D}(X_0, Y_0) = \frac{\sum_{x,y} I(x,y)I(x-Y_0, y-Y_0)}{\sum_{x,y} I^2(x,y)} \tag{1}$$

Each processed image is represented as an intensity distribution $I(x, y)$, where x and y are integer pixel coordinates. For each image, a copy of the same image is made and offset by the vector $\vec{R} = (X_0, Y_0)$, where the offsets X_0 and Y_0 are also in units of pixels. The overlapping portions of the two images are multiplied, and the product is normalized by the square of the image's total integrated pixel intensity, $\sum_{x,y} I^2(x, y)$, as shown in Fig. 4. Because the image is multiplied by a copy of itself, this form of correlation is known as *auto*-correlation; choosing two different images quantifies their degree of *cross*-correlation, but the underlying calculation is otherwise the same. When the offset is zero, the overlap is complete:

$$C_{2D}(0,0) \equiv 1$$

As the offset increases, the function decays to zero:

$$C_{2D}(\infty,\infty) \to 0$$

The resulting C_{2D} for the image in Fig. 3e is shown in

Fig. 5a. The azimuthal average of C_{2D} is the 1D autocorrelation function,

$$C_{1D}(R) \equiv \langle C_{2D}(X_0, Y_0)\rangle_\theta = \langle C_{2D}(R\cos\theta, R\sin\theta)\rangle_\theta \qquad (2)$$

where $R \equiv |\vec{R}| = \sqrt{X_0^2 + Y_0^2}$ is the scalar offset magnitude, and the brackets $\langle\rangle_\theta$ indicate an azimuthal average over all angles $\theta \equiv \arctan(Y_0/X_0)$. The characteristic length scale is the local maximum at R_{max}, which quantifies the average separation between bright features in the image [3] and is marked with a blue dot on the plot of C_{1D} in Fig. 5b. R_{max} grows over time to magnitudes comparable to the length L of the image. Using autocorrelation to locate the peak position in real space, when R_{max} is tens to hundreds pixels, is far more accurate than using a fast Fourier transform to locate the corresponding peak in reciprocal space at $2\pi L/R_{max}$ pixels. This value quickly shrinks to just a few pixels, requiring sub-pixel accuracy to locate the peak properly, and is far more susceptible to noise effects than the more robust real-space analysis.

4.1 Simple CPU-based MATLAB and C implementations

In the most straightforward algorithm to calculate C_{2D}, each pixel in the image at position (x, y) is multiplied by another pixel in the offset copy at $(x - X_0, y - Y_0)$, as shown in Fig. 4. Four loops are needed to cover all pixel values at all offsets. Our simple MATLAB code loops over X_0 and Y_0, then uses its native matrix operations to multiply the image by a copy offset by (X_0, Y_0) and sum the result in Eq. 1. We measure the time to calculate $C_{1D}(R)$ for the image in Fig. 3e on an Intel Core2 Quad 2.4 GHz CPU for all offsets up to R pixels, and plot these execution times with red symbols in Fig. 6. We obtain the same performance running the code from the internal MATLAB command line, and by compiling the program first and running it externally; MATLAB takes several hours to calculate $C_{1D}(250)$. Expressing the same algorithm (Fig. 4) using C requires four nested `for()` loops: two outer loops

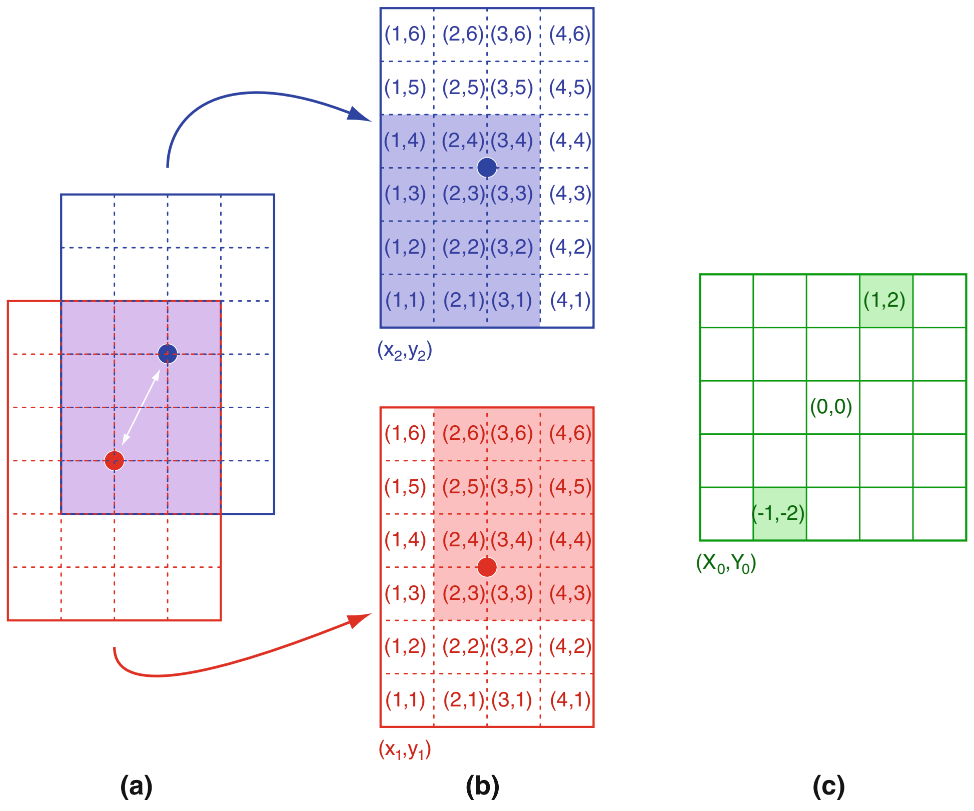

Fig. 4 Schematic of a simple algorithm to calculate the autocorrelation of an example image with dimensions of 4 × 6 pixels, at an offset of $(X_0, Y_0) = (+1, +2)$. **a** The original image is shown in red; the copy offset by $(+1, +2)$, in blue. The center of each image is marked by a colored circle. The overlapping area multiplied in the two images is shaded in purple. **b** In the blue image, the shaded overlap region corresponds to $1 \le x_2 \le 3$ and $1 \le y_2 \le 4$; in the red image, $2 \le x_1 \le 4$ and $3 \le y_1 \le 6$. All elements in these overlapping regions are multiplied and summed, following Eq. 1. In the images above, the pixel value at $(x_2, y_2) = (1, 1)$ is multiplied by $(x_1, y_1) = (1 + 1, 1 + 2) = (2, 3)$; $(x_2, y_2) = (1, 2)$ by $(x_1, y_1) = (1 + 1, 2 + 2) = (2, 4)$; and so on. The total sum of all products in the overlap region forms the numerator in Eq. 1; this value is stored in the $C_{2D}(X_0 = 1, Y_0 = 2)$ matrix element shaded in green in **c**. By symmetry, $C_{2D}(1, 2) = C_{2D}(-1, -2)$, since both the original image and the copy are identical

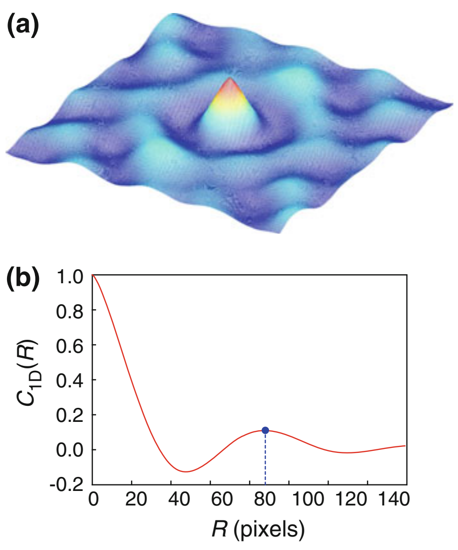

Fig. 5 **a** $C_{2D}(X_0, Y_0)$ calculated for the image shown in Fig. 3e. **b** $C_{1D}(R)$, calculated by azimuthally averaging C_{2D} shown in (a). The function decays from $C(0) = 1$ to a trough, and then rises to a secondary maximum at R_{max} = 88 pixels, which defines the characteristic length scale and is indicated by the blue dot

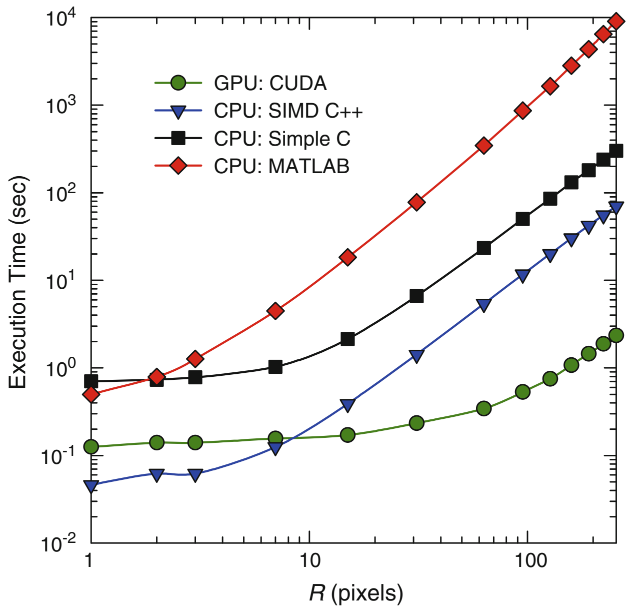

Fig. 6 Total program execution times to calculate $C_{1D}(R)$ for all offsets up to R pixels. CPU implementations using MATLAB, simple C and SIMD-optimized C++ are indicated with red diamonds, black squares and blue triangles, respectively; the GPU implementation using CUDA C is marked with green circles. At useful offsets of $R > 100$, the GPU requires seconds to calculate what requires hours in MATLAB on the CPU. In all cases, the azimuthal-average in Eq. 2 takes less than a second to calculate; the plotted data therefore essentially represents the time to calculate C_{2D}

over X_0 and Y_0, as before, and two inner loops over x and y [3] in place of the MATLAB matrix multiplication. This simple C implementation is about 30 times faster than MATLAB, as shown with black symbols in Figs. 6 and 7.

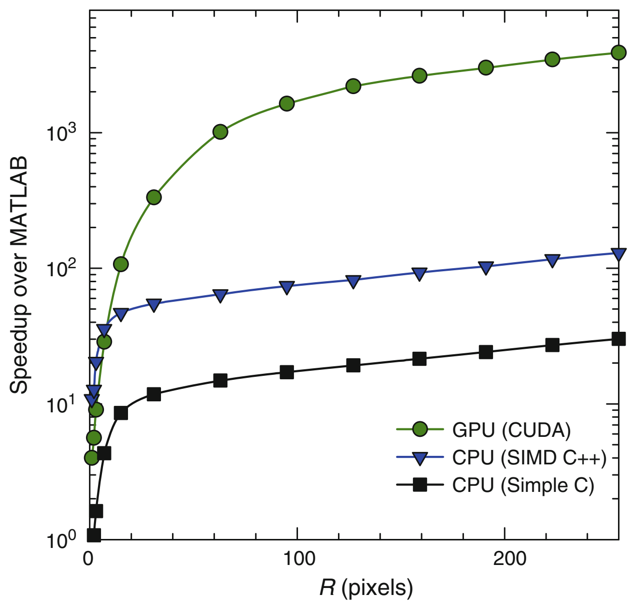

Fig. 7 Speedup of the different implementations relative to MATLAB. For $R > 250$, simple C is 30 times faster; optimized C++, 125 times faster; and CUDA C, 4000 times faster

4.2 Accelerated CPU-based C++ implementation optimizing cache and using SIMD

Within the innermost loop, the simple C code multiplies the values of pixels located at (x, y) and $(x - X_0, y - Y_0)$; in general, these two pixel values will reside in locations far away from each other in main memory, and will rarely be loaded into the CPU's memory cache at the same time. Direct multiplication thus requires two independent accesses of main memory, a relatively slow process in the absence of caching. We therefore reorder the loops to access adjacent memory locations consecutively, allowing the CPU to load the whole neighborhood of pixels into the cache once. In our re-ordered code, the outermost loop is over the Y_0, as before. The second loop is over y; the third, over X_0. With this arrangement, the innermost loop multiplies pixels with consecutive x coordinates; these reside in adjacent memory locations that are cached when the memory is accessed, significantly speeding memory access. We also exploit the symmetry $C_{2D}(X_0, Y_0) = C_{2D}(-X_0, -Y_0)$ to calculate only half of the Y_0 values; these changes improve performance by a factor of two.

Additional performance comes by carrying out the multiplication in Eq. 1 in parallel, since each product is independent of all others. Modern CPUs, such as Intel's Pentium and Core2, have specific circuitry devoted to vector operations, where a single instruction is executed on multiple data at the same time. SIMD instructions that exploit this special hardware can be added automatically to standard C++ code by modern compilers, such as the Intel Compiler 11 that we use. In a process called “autovectorization,” the compiler examines `for()` loops and adds appropriate parallel SIMD instructions if performance will improve as a result [5]. Without further changing the code, but only recompiling with the proper options, we gain an additional factor of two in performance as a result of the SIMD instructions. Overall, this optimized C++ code executes four times faster than the simple C implementation, and more than 100 times faster than the MATLAB implementation, as shown in Figs. 6 and 7.

4.3 Parallel GPU-based CUDA C implementation

To further improve performance with greater parallelism, we implement the autocorrelation using CUDA C and execute the code on an NVIDIA Tesla C1060, which contains a GT200 GPU [48]. The GT200 contains thirty streaming multiprocessor (SM) units; each SM contains eight streaming processors (SP) and 16 KB of high-speed shared memory accessible to all SPs within the same SM. Each of the 240 SPs is analogous to a simplified CPU core, and interacts with the 4 GB of GPU main memory in the form of generic read/write global memory, and cached read-only texture memory for large 2D arrays. The CPU exchanges data with global and texture memory over the PCI express (PCIe) bus, but cannot access any data in shared memory. Each SP executes one thread at a time; each SM executes one block of threads at a time, with its constituent threads exchanging data via shared memory [48].

We first implement the simple MATLAB/C algorithm of Fig. 4 using CUDA C, loading the image into texture memory. Each GPU thread is assigned a particular (X_0, Y_0) offset, and sums over x and y pixel coordinates with two `for()` loops. This simple GPU implementation executes about six times faster than the simple C approach, due to the large number of processors and high memory bandwidth inside the graphics card. However, this algorithm is only slightly faster than the optimized C++ CPU code, because it suffers from the same design limitation inherent in the simple C implementation: multiplication of two pixels at (x, y) and $(x - X_0, y - Y_0)$ requires reading from two distant locations in the GPU’s texture memory, which on average will not be cached. As a result, every pass through the inner loop in every thread incurs two memory-access delays of up to hundreds of clock cycles each; SPs, therefore, spend most of their time waiting for data to arrive instead of calculating.

To rectify this problem, we create a new CUDA C implementation which places pixels to be multiplied into shared memory. Because shared memory is so much smaller than global memory, we can only load a pair of lines of image data into shared memory at one time. As a result, we organize the calculation so that each thread sums the contribution of a single line in the image to a running total at each offset; thus, the final $C_{2D}(X_0, Y_0)$, stored in global memory, accumulates the results of many threads.

We use a two-dimensional grid of thread blocks, and a one-dimensional line of threads in each block. Each thread has three unique indices: block indices `blockIdx.x` and `blockIdx.y`, and thread index `threadIdx.x`. Each block loads two lines of the image into shared memory, whose y-coordinates are given by y_0 = `blockIdx.x` (line 9 in code Listing 1) and y_1 = `blockIdx.x + blockIdx.y` (line 10), as shown in Fig. 8a. Each thread contains a single `for()` loop over the x coordinate of the image lines; inside this loop, the products of the pixel values at x and $x - X_0$ are summed (lines 14–16). The same value of the pixel at x is broadcast to all threads in the block simultaneously, while no two threads access the same memory address for the pixel at $x - X_0$, because each thread has a different X_0 = `threadIdx.x` (line 15); this strategy reduces shared-memory queues and memory-bank conflicts, as shown in Fig. 8b. After performing this sum, each thread adds its contribution from line y to the accumulating sum collected for the offset at (X_0, Y_0), as shown in Fig. 8c. We use an in-place atomic addition operation to increment directly the value stored in the corresponding element of the global memory array `output_2DCorr` (line 18); this obviates the need to read the old value from global memory to an SP, add the new result and write it back.

```
__global__ void AutoCorr_OneLine(int *output_2DCorr, int
    image_width, int max_offset) {
   __shared__ float ImageDataLine1[IMAGE_WIDTH];
   __shared__ float ImageDataLine2[IMAGE_WIDTH + OFFSET_SIZE];
   ImageDataLine2[threadIdx.x] = 0;
   ImageDataLine2[image_width+2+threadIdx.x] = 0;
   __syncthreads();
   int num_horiz_blocks=__float2int_ru(__int2float_rd(image_width)/
       __int2float_rd(blockDim.x)), n = 0;
   for(n=0; n<num_horiz_blocks; n++) {
      ImageDataLine1[__umul24(n,blockDim.x)+threadIdx.x] = tex2D(
          inputTexture, __umul24(n, blockDim.x) + threadIdx.x,
          blockIdx.x + 1);
      ImageDataLine2[max_offset+__umul24(n,blockDim.x) + threadIdx.
          x] = tex2D(inputTexture, __umul24(n, blockDim.x) +
          threadIdx.x, blockIdx.x + blockIdx.y + 1);
   }
   __syncthreads();
   float linesum = 0;
   for(int x_coord = 0; x_coord < image_width; x_coord++) {
      linesum += ImageDataLine1[x_coord] * ImageDataLine2[x_coord+
          threadIdx.x];
   }
   int coeff_out = __float2int_rn(100.0 * linesum / __int2float_rd
       ((image_width - abs((int) threadIdx.x - max_offset))));
    int temp = atomicAdd(&output_2DCorr[threadIdx.x + __umul24((2*
        max_offset + 1), blockIdx.y)], coeff_out);
}
```

Listing 1 CUDA C code for the kernel using shared memory. The function has several arguments: a pointer to the global memory array `output_2DCorr`, where the output data is stored; `image_width`, the width of the image in pixels; and `max_offset`, the maximum offset up to which the correlations are calculated. Two constants, `IMAGE_WIDTH` and `OFFSET_SIZE`, are macros defined to be 800 and 514, respectively, and represent maximum possible values for image width and offset in units of pixels; these must be explicitly defined in the code, because CUDA C does not at present allow dynamic memory allocation in shared memory. We launch this kernel with the configuration `threads2(2* max_offset + 1, 1)` and `grid2(image_height-max_offset+2,max_offset)`

Together, these optimizations dramatically increase the speed of the code: the execution time, even for the largest offsets, is mere seconds; by contrast, MATLAB requires hours, as shown in Fig. 6. Our optmized CUDA C implementation runs nearly 4,000 times faster than MATLAB, 130 times faster than simple C, and 30 times faster than the SIMD-optimized C++, as shown in Fig. 9. All of these codes yield the same results, up to small rounding errors, and demonstrate a linear growth of $R_{\max}(t)$ over time t [13], as shown in Fig. 10.

5 Discussion

Our method to calculate autocorrelation rapidly joins a family of related algorithms that tackle different aspects of the general problem of correlation. An optimized method to calculate a single matrix multiplication, computationally equivalent to the autocorrelation at zero offset, is presented as an example chapter in the NVIDIA CUDA programming guide. There, large matrices are broken into a number of smaller two-dimensional submatrices, called tiles, which are loaded into shared memory; a block of threads performs the multiplication of two tiles, then these partial sums are added up for the complete multiplication results (see, for example, examples and tutorials at [48]). These algorithms operate efficiently because the tiles are two-dimensional and fit entirely into shared memory; the actual multiplication, which requires two nested loops in sequential code, is carried out by a two-dimensional block of threads. The size of the tiles is determined by the capabilities of the GPU, specifically the number of threads in a block, and the total shared memory size.

Our algorithm, however, is not merely a simple extension of these principles. We have four nested loops, which exceeds the total dimensions available to threads in a block. That is, we cannot simply replace each loop with a thread coordinate, and directly extend these well-known approaches. Instead, we use the described line-by-line approach, and replace the four loops with two grid coordinates, one thread coordinate, and one loop inside each thread. This architectural choice was motivated by the sizes of our data and the capabilities of the current generation of NVIDIA GPUs. Our images are roughly a thousand pixels on a side; therefore, several lines will fit into the 16 kB of shared memory available on the GT200. Moreover, we typically examine offsets of a couple of hundred pixels, so that setting the number of threads to equal twice the maximum offset conveniently falls within the limitation of the GT200's

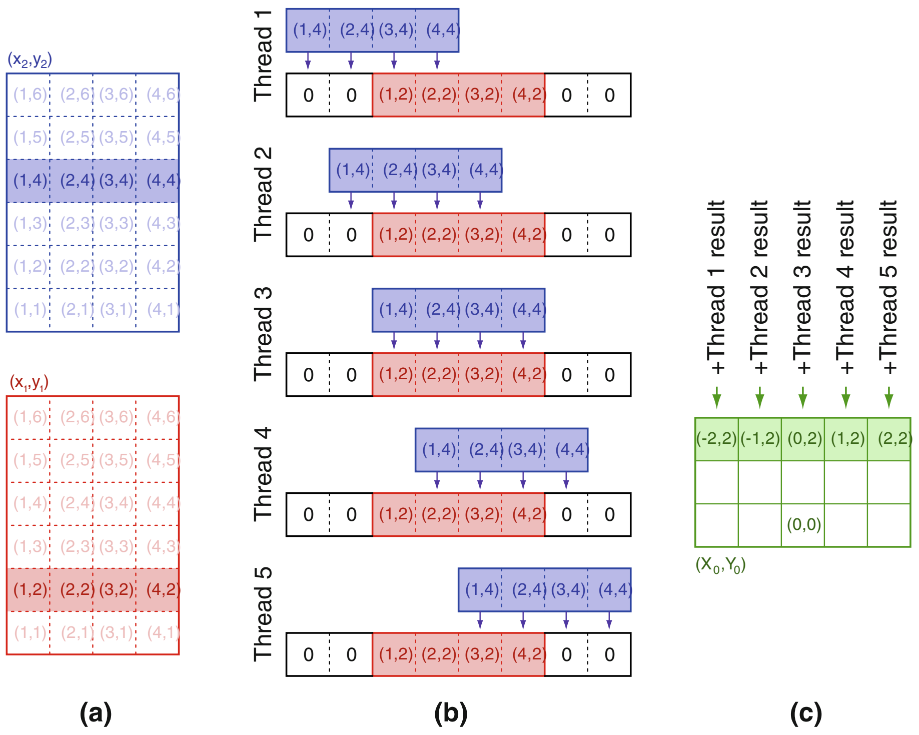

Fig. 8 Schematic of a CUDA algorithm using shared memory to calculate the autocorrelation of the image in Fig. 4a for $-2 < X_0 < 2$ and $Y_0 = 2$. **a** The CUDA algorithm loads into shared memory only one line from the original image at $y_1 = 2$, shown shaded in red; and one line from the offset copy at $y_2 = y_1 + Y_0 = 4$, shown shaded in blue. **b** Each thread multiples the same two lines in shared memory, but with a different offset equal to its index, X_0 = `threadIdx.x`. For example, thread 1 calculates the sum of the products $I(1, 4) \times 0 + I(2, 4) \times 0 + I(3, 4) \times I(1, 2) + I(4, 4) \times I(2, 2)$, as shown in the top line. **c** the results of the partial sums calculated by each thread are added to the accumulating values for $C_{2D}(X_0, Y_0)$ in global memory

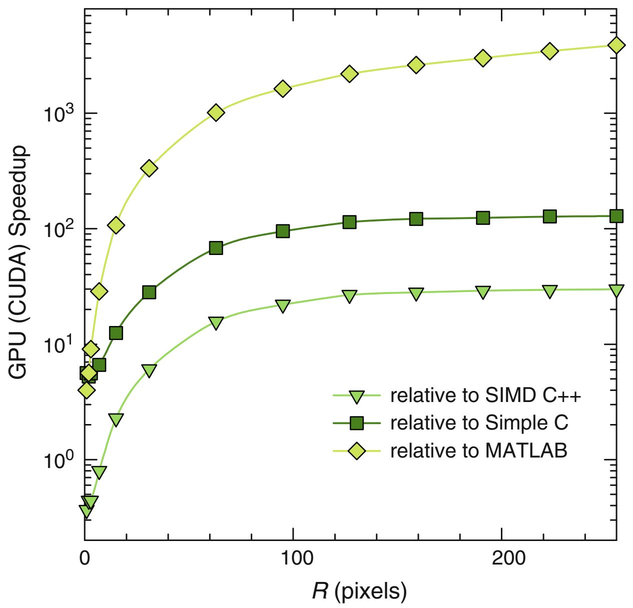

Fig. 9 Speedup of the GPU-based CUDA C code relative to the CPU-based implementations. For $R > 250$, the CUDA C implementation is 30 times faster than SIMD-optimized C++, 130 times faster than simple C, and 4000 times faster than MATLAB

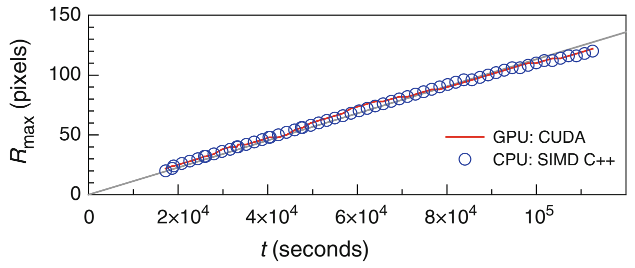

Fig. 10 Growth of R_{max} as a function of time t. The results from CPU-based MATLAB/C/C++ (*blue circles*) codes are the same as from the GPU-based CUDA C implementation (*red line*), up to small rounding differences. Both sets of data conform to a best-fit black line that passes through the origin, demonstrating a linear growth of the sample's characteristic length scale, a hallmark of a type of phase separation known as late-stage spinodal decomposition [13].

maximum of 512 threads per block. By calculating the sums for multiple offsets in each thread block, we maximize the usage of our lines of image data residing in shared memory. In our particular situation, there is no obvious reason to use square tiles: we would rapidly run out of shared memory for any useful offset value, as the tile size grows as the square of the offset value following our approach. Using a small tile in shared memory, however, might be useful in the calculation of particle image velocimetry (PIV) or optical flow, as these procedures correlate smaller subregions of images, instead of the entire large images in our case. Other types of correlation on the GPU, for instance of a one-dimensional time series, or the two-dimensional angular distribution of the cosmic microwave background radiation [32], require approaches that differ significantly from ours.

The performance of our CUDA C code exceeds that of the simple C code, executed on a recent Intel Core2 Quad CPU, by more than two orders of magnitude. This performance increase is relatively rare among CUDA applications involving real experimental data [16]; more typical maximum speedups range from 30 to 60 times [19, 26, 34, 42]. Several factors limit GPU-based analysis of real data. Data transfers from CPU host memory to the GPU must cross the relatively slow PCIe bus. Moreover, data size or format is often ill-suited for the GPU memory organization; instead, data must be partitioned into pieces suitable for analysis, adding overhead. By contrast, simulations and numerical calculations remove these constraints; they generally transport little input data across the PCIe bus, and they can format their generated data (e.g. random numbers) optimally. As a result, the best CUDA simulation speedups are between 2 to 3 orders of magnitude faster than implementations on the older Pentium architecture [1, 18, 45]; however, top speedups of 100 times relative to the more recent Core2 CPU are more common [4, 15, 41]. Simulations also benefit from community expertise in programming and algorithm design; with few exceptions [1], papers describing CUDA simulation speedups of two or more orders of magnitude have corresponding authors in computer engineering departments [4, 15, 18, 41, 45]. By contrast, experimentalists have on average far less programming experience. Many experimental papers achieving the best CUDA speedups are collaborations involving co-authors in computing-related departments [16, 26, 34, 42], with few exceptions [19].

Is CUDA practical for scientists and engineers who do not specialize in programming? This dilemma reflects the tradeoff between performance and development effort. In general, solutions that require the least programming tend to execute slowest. MATLAB programs are quite easy to write, but their performance is heavily constrained by the significant inefficiencies and overheads inherent in the MATLAB application, which cannot be removed by the programmer. For $R \approx 100$, our CUDA C code executes completely in the same time that MATLAB merely opens the image and loads it into memory, with no calculation of Eq. 1 at all. No matter how fast that calculation runs—even with specialized toolboxes, multiprocessor support, or a cluster of machines—the MATLAB code will always trail by orders of magnitude in this particular application.

Using C/C++ requires somewhat greater development effort, but boosts performance significantly. Application overheads are minimized, and optimizing compilers and libraries can automatically utilize SIMD instructions without much additional programmer effort [5]. A major motivation for our inclusion of the autovectorized C++ code was to demonstrate that, by only switching the order of a few loops and recompiling with different options, performance can be improved by half an order of magnitude. Undoubtedly, even greater gains could be achieved with explicitly adding CPU assembler or intrinsics, but require a substantially greater amount of skill and effort.

In between CPU-based C++ and a full CUDA C implementation are ordinary C libraries available for some common operations that execute on the GPU, including matrix multiplication (CuBLAS), fast Fourier transforms (CuFFT) and a number of image processing functions (NPP) [48]. The inner workings of these functions on the GPU are transparent to the programmer, who only interacts with the standard C/C++ interface; some scientific applications leverage GPGPU capabilities in this way [38]. However, because only some operations in an otherwise serial program can execute on the GPU, total speedups exceeding a factor of ten are rare with this approach.

To achieve speedups greater than an order of magnitude, some specific programming of the GPU's hardware is usually required; this is a major difference between programming the GPU and the CPU. In general, MATLAB and C/C++ do not require, or even allow, direct access to the CPU hardware and memory caches. Indeed, this access is usually not necessary: the x86 CPU architecture is mature, and optimizing compilers will typically generate faster code than that tuned by hand in most cases [5]. By contrast, the GPU architecture is still evolving. Not only are the compilers at an earlier stage of development, but the optimizations of parallel algorithms also require the interplay of many more factors, many based on the specifics of the application that cannot simply be guessed by a compiler. Consequently, the GPU programmer must exercise a greater degree of control over the hardware to maximize performance. The earliest GPGPU applications required programming the graphics pipeline explicitly [30] and manually inserting processor-specific assembler instructions. In particular situations, this approach can still lead to impressive benefits today on GPUs [42] and the closely-related Cell processor inside the Sony Playstation3 [26, 46], but requires tour-de-force programming feats. Fortunately, CUDA opens up the GPU architecture for general computation in a way that requires no previous knowledge of graphics or assembler. CUDA programming does require a reasonable foundation in C/C++, though, and a willingness to redesign algorithms to take advantage of specific features of current GPU hardware, particularly the various sizes and speeds of the different memory buffers. Nevertheless, our work uses a few dozen lines of CUDA C code to achieve thousands-fold performance increases relative to MATLAB, demonstrating the practicality of CUDA in a real-world science application.

In general, however, such large performance increases may not always be achievable. Our algorithm's speed is due to several factors that make it a nearly-ideal candidate for CUDA implementation. We perform a large number of calculations on a moderately-sized data set, so that the GPU spends far more time performing numerical operations than transferring data; this is a general principle which all of the most efficient algorithms use. We use the line-by-line approach to maximize the usage of shared memory: different offsets are all calculated within the same thread block, maximizing the amount of computation performed on data already in shared memory; in general, maximizing shared memory usage may be the single most important factor in achieving high CUDA performance. There is a cost in flexibility to this specific approach, however, as offsets in our algorithm are limited by the maximum number of threads that can occur in a single block, 512 on the current GT200 hardware [48]. This size happens to be convenient for our specific application, where the image size is limited by the resolution of the CCD chip in the cameras on-board the ISS; correlations greater than a third of the image width suffer from poor statistics, so we do not need to measure offsets greater than about 250 pixels. Were our images an order of magnitude larger or smaller, we would have opted for a different approach. Finally, our algorithm repeatedly performs very simple operations, essentially only multiplication and addition, so that the GPU can be very effectively deployed; had our algorithm required more complex functions, or even a large number of divisions, the speedups would not have been so large relative to CPU-based code. Consequently, speedups exceeding two orders of magnitude may not be achievable in many applications, but most well-implemented CUDA algorithms can perform faster than CPU-based code by more than a factor of ten.

Tantalizingly, increasing performance by several orders of magnitude may enable qualitatively new applications in science and technology. While researchers often focus on minimizing development time, writing the simplest piece of software that will “get the job done” regardless of performance, the dramatic speedups offered by GPGPU may fundamentally change what that “job” is. Dramatically faster analysis can enable commensurate increases in the amount of data collected, particularly in fields like radio astronomy [16] where data collection is limited by processing rates downstream; observation of structures in space with increased resolution facilitates more-detailed understanding of how the universe formed. On a practical level, a patient often waits idly in an MRI machine during the reconstruction of a scan to check its accuracy; faster reconstruction yields more time to collect higher-resolution data [42] that may facilitate earlier detection of diseases like cancer. For our code, the frame rate obviously depends on image size; for standard video-sized images of 640×480, calculation of the autocorrelation of offsets up to 16 pixels will exceed real-time video rates of 30 frames per second. Beyond our particular application, our code could find real-time usage in stabilizing or tracking full-frame displacements in live-motion video, for example, bringing data analysis into the interactive real-time realm. In this regime, fully-analyzed results can be obtained fast enough to influence the data collection process itself [23], and existing applications include the real-time tracking of moving targets like faces [21]. Clearly, CUDA applications are beginning to have a significant impact in several key areas of science and engineering, and will undoubtedly continue to do so in the future.

Acknowledgments This work was supported by NASA grant NNX08AE09G and the NVIDIA professor partnership program. We thank A. Bik, J. Curley, A. Ghuloum, D. Luebke, E. Phillips, B. Saar, H. Saito, M. Schnubbel-Stutz, L. Vogt, and many helpful individuals throughout NASA and its contractors.

References

1. Alerstam, E., Svensson T., Andersson-Engels, S.: Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. JBO Lett. **13**, 060504 (2008). doi:10.1117/1.3041496
2. Anderson, J.A., Lorenz, C.D., Travesset, A.: General purpose molecular dynamics simulations fully implemented on graphics processing units. J. Comput. Phys. **227**, 5342–5359 (2008). doi: 10.1016/j.jcp.2008.01.047
3. Bailey, A.E., Poon, W.C.K., Christianson, R.J., Schofield, A.B., Gasser, U., Prasad, V., Manley, S., Segre, P.N., Cipelletti, L., Meyer, W.V., Doherty, M.P., Sankaran, S., Jankovsky, A.L., Shiley, W.L., Bowen, J.P., Eggers, J.C., Kurta, C., Lorik, Jr., T., Pusey, P.N., Weitz, D.A.: Spinodal decomposition in a model colloid–polymer mixture in microgravity. Phys. Rev. Lett **99**, 205701 (2007). doi:10.1103/PhysRevLett.99.205701
4. Belleman, R.G., Bédorf, J., Portegies Zwart, S.F.: High performance direct gravitational *N*-body simulations on graphics processing units II: an implementation in CUDA. New Astron. **13**, 103–112 (2008). doi:10.1016/j.newast.2007.07.004
5. Bik, A.J.C.: The Software Vectorization Handbook. Intel, Hillsboro (2004)
6. Bodnár, I., Dhont J.K.G., Lekkerkerker, H.N.W.: Pretransitional phenomena of a colloid polymer mixture studied with static and dynamic light scattering. J. Chem. Phys. **100**, 19614–19619 (1996)
7. Bodnár, I., Oosterbaan, W.D.: Indirect determination of the composition of the coexisting phases in a demixed colloid polymer mixture. J. Chem. Phys. **106**, 7777–7780 (1997)
8. Castaño-Díez, D., Mozer, D., Schoenegger, A., Pruggnaller S., Frangakis, A.S.: Performance evaluation of image processing algorithms on the GPU. J. Struct. Biol. **164**, 153–160 (2008). doi: 10.1016/j.jsb.2008.07.006
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Skadron, K.: A performance study of general-purpose applications on graphics processors using CUDA. J. Parallel Distrib. Comput. **68**, 1370–1380 (2008). doi:10.1016/j.jpdc.2008.05.014
10. Christiansen, M.: Adobe After Effects 7.0 Studio Techniques. Peachpit, Berkeley (2006)
11. Fernando, R., Kilgard, M.J.: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley, Boston (2003)
12. Fraser F., Schewe, J.: Real World Camera Raw with Adobe Photoshop CS3. Peachpit, Berkeley (2008)
13. Furukawa, H.: A dynamic scaling assumption for phase separation. Adv. Phys. **34**, 703–750 (1985)

14. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel Computing Experiences with CUDA. IEEE Micro **28**, 13–27 (2008)
15. Gumerov, N.A., Duraiswami, R.: Fast multipole methods on graphics processors. J. Comput. Phys. **227**, 8290–8313 (2008). doi:10.1016/j.jcp.2008.05.023
16. Harris, C., Haines K., Staveley-Smith, L.: GPU accelerated radio astronomy signal convolution. Exp. Astron. **22**, 129–141 (2008). doi:10.1007/s10686-008-9114-9
17. Ibrahim, K.Z., Bodin, F., Pène, O.: Fine-grained parallelization of lattice QCD kernel routine on GPUs. J. Parallel Distrib. Comput. **68**, 1350–1359 (2008). doi:10.1016/j.jpdc.2008.06.009
18. Li, H., Kolpas, A., Petzold, L., Moehlis, J.: Parallel simulation for a fish schooling model on a general-purpose graphics processing unit. Concurr. Comput. Pract. Exp. (2008). doi:10.1002/cpe.1330
19. Liu, S., Li, P., Luo, Q.: Fast blood flow visualization of high-resolution laser speckle imaging data using graphics processing unit. Opt. Express **16**, 14321–14329 (2008). doi:10.1364/OE.16.014321
20. Liu, W., Schmidt, B., Voss, G., Müller-Wittig, W.: Accelerating molecular dynamics simulation using Graphics Processing Units with CUDA. Comp. Phys. Comm. **179**, 634–641 (2008). doi:10.1016/j.cpc.2008.05.008
21. Lozano, O.M., Otsuka, K.: Real-time Visual Tracker by Stream Processing. J. Signal Process. Syst. (2008). doi:10.1007/s11265-008-0250-2
22. Lu, P.J., Conrad, J.C., Wyss, H.M., Schofield, A.B., Weitz, D.A.: Fluids of Clusters in Attractive Colloids. Phys. Rev. Lett. **96**, 028306 (2006). doi:10.1103/PhysRevLett.96.028306
23. Lu, P.J., Sims, P.A., Oki, H., Macarthur, J.B., Weitz, D.A.: Target-locking acquisition with real-time confocal (TARC) microscopy. Opt. Express **15**, 8702–8712 (2007). doi:10.1364/OE.15.008702
24. Lu, P.J., Zaccarelli, E., Ciulla, F., Schofield, A.B., Sciortino, F., Weitz, D.A.: Gelation of particles with short-range attraction. Nature **453**, 499–503 (2008). doi:10.1038/nature06931
25. Lu, P.J.: Gelation and Phase Separation of Attractive Colloids. Harvard University Ph.D. Thesis (2008)
26. Manavski, S.A., Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith–Waterman sequence alignment. BCM Bioinf. **9**(Suppl 2), S10 (2008). doi:10.1186/1471-2105-9-S2-S10
27. Marziale, L., Richard III, G.C., Roussev, V.: Massive threading: Using GPUs to increase the performance of digital forensics tools. Digital Investigation **4S**, S73–S81 (2007). doi:10.1016/j.diin.2007.06.014
28. McCool, M., Du Toit, S.: Metaprogramming GPUs with Sh. Peters, Wellesley (2004)
29. Nguyen, H. (ed.): GPU Gems 3. Addison-Wesley, Upper Saddle River (2007)
30. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A.E., Purcell, T.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**, 80–113 (2007)
31. Pharr, M. (ed.): GPU Gems 2. Addison-Wesley, Upper Saddle River (2005)
32. Roeh, D.W., Kindratenko V.V., Brunner, R.J.: Accelerating cosmological data analysis with graphics processors. In Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units. ACM, Washington (2009)
33. Ruiz, A., Ujaldon, M., Cooper, L., Huang, K.: Non-rigid Registration for Large Sets of Microscopic Images on Graphics Processors, J. Sign. Process. Syst. (2008) doi:10.1007/s11265-008-0208-4
34. Samant, S.S., Xia, J., Muyan-Özçelik, P., Owens, J.D.: High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy. Med. Phys. **35**, 3546–3553 (2008). doi:10.1118/1.2948318
35. Schatz, M.C., Trapnell, C., Delcher, A.L., Varshney, A.: High-throughput sequence alignment using Graphics Processing Units. BCM Bioinformatics **8**, 474 (2007). doi:10.1186/1471-2105-8-474
36. Schenk, O., Christen, M., Burkhart, H.: Algorithmic perfomance studies on graphics processing units. J. Parallel Distrib. Comput. **68**, 1360–1369 (2008). doi:10.1016/j.jpdc.2008.05.008
37. Shimobaba, T., Ito, T., Masuda, N., Abe, Y., Ichihashi, Y., Nakayama, H., Takada, N., Shiraki, A., Sugie, T.: Numerical calculation library for diffraction integrals using the graphic processing unit: the GPU-based wave optics library. J. Opt. A: Pure Appl. Opt. **10**, 075308 (2008). doi:10.1088/1464-4258/10/7/075308
38. Shimobaba, T., Sato, Y., Miura, J., Takenouchi, M., Ito, T.: Real-time digital holographic microscopy using the graphics processing unit. Opt. Express **16**, 11776–11781 (2008). doi:10.1364/OE.16.011776
39. Sintorn, E., Assarsson, U.: Fast parallel GPU-sorting using a hybrid algorithm. J. Parallel Distrib. Comput. **68**, 1381–1388 (2008). doi:10.1016/j.jpdc.2008.05.012
40. Stantchev, G., Dorland W., Gumerov, N.: Fast parallel Particle-To-Grid interpolation for plasma PIC simulations on the GPU. J. Parallel Distrib. Comput. **68**, 1339–1349 (2008). doi:10.1016/j.jpdc.2008.05.009
41. Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., Schulten, K.: Accelerating Molecular Modeling Applications with Graphics Processors. J. Comput. Chem. **28**, 2618–2640 (2007). doi:10.1002/jcc.20829
42. Stone, S.S., Haldar, J.P., Tsao, S.C., Hwu, W.-m.W., Sutton, B.P., Liang, Z.-P.: Accelerating advanced MRI reconstructions on GPUs. J. Parallel Distrib. Comput. **68**, 1307–1317 (2008). doi:10.1016/j.jpdc.2008.05.013
43. Taylor, S.: Intel Integrated Performance Primitives. Intel, Hillsboro, OR (2004)
44. Thibault, J.C., Senocak, I.: CUDA Implementation of a Navier–Stokes solver in multi-GPU desktop platforms for incompressible flows. In 47th AIAA Aerospace Sciences Meeting and Exhibit (2009)
45. Van Meel, J.A., Arnold, A., Frenkel, D., Portegies Zwart, S.F., Belleman, R.G.: Harvesting graphics power for MD simulations. Mol. Simulation **34**, 259–266 (2008). doi:10.1080/08927020701744295
46. Wirawan, A., Kwoh, C.K., Hieu, N.T., Schmidt, B.: CBESW: sequence alignment on the Playstation 3. BCM Bioinf. **9** 377 (2008). doi:10.1186/1471-2105-9-377
47. Zaccarelli, E., Lu, P.J., Ciulla, F., Weitz, D.A., Sciortino, F.: Gelation as arrested phase separation in short-ranged attractive colloid-polymer mixtures. J. Phys. Condens. Matter **20**, 494242 (2008). doi:10.1088/0953-8984/20/49/494242
48. http://www.nvidia.com/cuda
49. http://www.khronos.org/opencl

Author Biographies

Peter J. Lu received his AB *summa cum laude* in physics (2000) from Princeton University, and AM (2002) and PhD (2008) in physics from Harvard University. He is presently a post-doctoral research fellow in the department of physics and in the school of engineering and applied sciences at Harvard University; his main focus is on the physics of attractive colloids and the integration of high-performance imaging and analysis techniques. He has developed PLuTARC, a real-time target-locking system for use in confocal microscopy, allowing the observation of freely-moving objects, such as biological cells, for extended periods of time. He has also published the discoveries of modern quasicrystal geometry in medieval Islamic architectural tilings; the first precision compound machines, from ancient China; the first use of diamond, in prehistoric China; and the first quasicrystalline mineral found in nature.

Hidekazu Oki received his BSE in computer science (2000) and M.Eng in electrical engineering (2003) from Princeton University. He currently works as a software engineer, focusing on cryptography and security in smart-card software development, at Dai Nippon Printing in Tokyo. He collaborates in the implementation and improvement of scientific computing software, using computer architecture-aware software-performance optimizations. His past work includes data-cache-aware optimization and software optimization using SSE instructions.

Catherine A. Frey received her degree in physics and astronomy from the Bowling Green State University, in Ohio. She currently works at Zin Technologies, where she has integrated scientific payloads for the Space Shuttle and International Space Station, for the NASA Glenn Research Center, for the past 19 years. She has trained astronauts and participated in real-time operations of flight payloads since 2001.

Gregory E. Chamitoff received his BS in electrical engineering (1984) from California Polytechnic State University, an MS (1985) in aeronautical engineering from the California Institute of Technology, a PhD in aeronautics and astronautics (1992) from the Massachusetts Institute of Technology, and an MS in space science/planetary geology (2002) from the University of Houston Clear Lake. While at MIT and Draper Labs (1985–1992), he performed stability analysis for the deployment of the Hubble Space Telescope, designed flight control upgrades for the Space Shuttle autopilot, and developed attitude control system software for the Space Station. In 1995, he joined Mission Operations at the Johnson Space Center, where he developed software applications for spacecraft attitude control monitoring, prediction, analysis, and maneuver optimization, including the 3D 'big screen' display of the Station and Shuttle used by Mission Control. He served 6 months aboard the ISS as Expedition 17–18 ISS Flight Engineer and Science Officer.

Leroy Chiao received a BS in chemical engineering (1983) from the University of California, Berkeley, and an MS (1985) and PhD (1987) in chemical engineering from the University of California, Santa Barbara. He worked for Hexcel until 1989, on a joint NASA-JPL/Hexcel project to develop an optically correct, polymer composite precision segment reflector, for space telescopes, and on cure modeling and finite element analysis. In 1989, he began work at the Lawrence Livermore National Laboratory on processing research for fabrication of filament-wound and thick-section aerospace composites. An instrument-rated pilot, he flew three times aboard the Space Shuttle. On the STS-65 Columbia mission (July 8–23, 1994) he became the 196th NASA Astronaut to fly in space and the 311th human in space. During 9-day the STS-72 Endeavour (January 11–20, 1996) mission, he performed two spacewalks designed to demonstrate tools and techniques to be used in the assembly of the International Space Station, and became the first Asian-American and ethnic Chinese to perform a spacewalk. He was the EVA/Construction lead for the STS-92 Discovery (October 11–24, 2000) mission, which continued ISS assembly and prepared the ISS for its first resident crew. From October 13, 2004 to April 24, 2005, he commanded ISS Expedition 10 and served as NASA Science Officer, performing 20 science experiments and two repair and installation space walks, becoming the first Asian-American and ethnic Chinese Mission Commander of the ISS.

Edward M. Fincke attended the Massachusetts Institute of Technology on an Air Force ROTC scholarship and received a BS (1989) in aeronautics and astronautics, and a BS in earth, atmospheric and planetary sciences. He received an MS in aeronautics and astronautics (1990) from Stanford University, an AS in earth sciences/geology (1993) from El Camino College, and an MS in physical sciences/planetary geology (2001) from the University of Houston, Clear Lake. After graduation from MIT in 1989, he attended a summer exchange at Moscow Aviation Institute in the former Soviet Union, where he studied cosmonautics. After graduation from Stanford University in 1990, he entered the United States Air Force as a space systems engineer and a space test engineer at Los Angeles Air Force Base; and as a flight test engineer at Edwards and Eglin Air Force Bases, flying in F-16 and F-15 aircraft and attaining the rank of Colonel. In 1996, he was the United States flight test liaison to the Japanese/United States XF-2 fighter program at the Gifu Air Base in Japan. At NASA from 1996, he has served as ISS spacecraft communicator, a member of the crew test support team in Russia, back-up crewmember for ISS expeditions 4 and 6, and back-up commander for ISS expeditions 13 and 16. He has flown aboard two ISS missions. During ISS Expedition 9 (April 18–October 23, 2004), he served as NASA science officer and flight engineer, where he conducted numerous experiments, maintained ISS systems, and performed four space-walks. He commanded Expedition 18 (12 October 2008–8 April 2009), which helped prepare the station for future 6-person crews and hosted the Space Shuttle crews of STS-126 and STS-119.

C. Michael Foale received a BA in physics (1978), completing the natural sciences tripos with first-class honors, and his DPhil in laboratory astrophysics (1982) from Queen's College, Cambridge. In June 1983, Foale joined NASA Johnson Space Center as payload officer in the Mission Control Center, coordinating payload operations on Space Shuttle missions STS-51G, STS-51-I, STS-61-B and STS-61-C. He has flown on four Space Shuttle missions as a crew member. STS-45 (March 24 to April 2, 1992) was the first of the ATLAS series of missions to study the atmosphere and solar interactions. STS-56 (April 9–17, 1993) carried ATLAS-2 and the SPARTAN retrievable satellite that made observations of the solar corona. STS-63 (February 2–11, 1995) was the first rendezvous with the Russian Space Station Mir. During STS-63 he made his first space walk, evaluating extremely cold spacesuit conditions, and exploring mass handling of the 2800-pound Spartan satellite. On May 15, 1997, he joined the crew aboard the Russian Space Station Mir, where he initially conducted science experiments, and later helped reestablish the Mir after it was degraded by a collision and depressurization. From December 19 to 27, 1999, he flew on STS-103 to repair and upgrade the Hubble Space Telescope, replacing the telescope's main computer and fine guidance sensor during an EVA. From October 20, 2003 to April 29, 2004, he commanded Expedition 8 aboard the International Space Station, and conducted numerous science experiments.

Sandra H. Magnus received a BS (1986) in physics and an MS in electrical engineering (1990) from the University of Missouri-Rolla, and a PhD in materials science and engineering (1996) from the Georgia Institute of Technology. From 1986–1991, she developed stealth technology at the McDonnell Douglas Aircraft Company, and also contributed to the propulsion system of the Navy's A-12 attack aircraft. She joined NASA in 1996, working initially in the astronaut office, in the payloads and habitability branches. From October 7–18, 2002 she flew aboard the Space Shuttle Atlantis during the STS-112 mission, operating the Shuttle's robotic arm during three spacewalks to deliver and install the third piece of the ISS's integrated truss structure. From November 2008 to March 2009, She served as the NASA science officer and second flight engineer aboard the ISS during Expedition 18, where she prepared the ISS to begin 6-person crew operations, supported two Orlan-based spacewalks and conducted a number of science experiments.

William S. McArthur Jr. received a BS in applied science and engineering (1973) from the US Military Academy, West Point, and an MS in aerospace engineering (1983) from the Georgia Institute of Technology. Commissioned as a second lieutenant in the US army following his West Point graduation, he was the top graduate in his flight class from the US Army Aviation School in 1976, and subsequently served as an aeroscout team leader and brigade aviation section commander with the 2nd Infantry Division in Korea. In 1978, he served as company commander, platoon leader, and operations officer with the 24th Combat Aviation Battalion in Savannah, Georgia. After graduating from Georgia Tech, he joined the faculty of West Point as assistant professor in the Department of Mechanics. In 1987, he graduated from the US Naval Test Pilot School and was designated an experimental test pilot. Joining NASA in 1990, he became an astronaut in 1991, and flew four space missions. During STS-58 (October 18–November 1, 1993), he performed extensive medical tests on himself, in addition to a number of other medical experiments, and made extensive radio contact with school children and amateur radio operators around the world. The STS-74 (November 12–20, 1995) mission was the second to dock with Russian space station Mir, during which he attached at permanent docking collar, transferred large payloads, and conducted experiments. During STS-92 (October 11–24, 2000) performed several space walks to assemble and configure several new structural elements of the ISS, preparing it to receive its first resident crew. From September 20, 2005 to April 8, 2006, he commanded ISS Expedition 12, and served as science officer. In addition to conducting a number of experiments, he was the first to dock to every Russian docking port on the ISS, and to conduct spacewalks with both US and Russian spacesuits.

Daniel M. Tani received his BS (1984) and MS (1988) in mechanical engineering from the Massachusetts Institute of Technology. In 1988, he joined Orbital Sciences Corporation, where he managed the mission and deployment of the Transfer Orbit Stage during the STS-51 space shuttle mission in 1993, before leading the development of launch procedures for the Pegasus unmanned rocket. Joining NASA in 1996, he flew aboard STS-108, the 12th shuttle flight to the ISS, where he assisted the installation of the Raffaello multi-purpose logistics module, and performed a space walk to wrap thermal blankets around the ISS solar array gimbals. On his second spaceflight, he served as the Expedition 16 (October 23, 2007 to February 20, 2008) flight engineer aboard the ISS, during which he performed robotic installation operations, and conducted numerous scientific experiments.

Peggy A. Whitson received her BS in biology/chemistry (1981) from Iowa Wesleyan College, and her PhD in biochemistry (1985) from Rice University, continuing as a post-doctoral fellow at Rice until 1986, before moving to NASA. From 1989 to 1993, She worked as a research biochemist at NASA-JSC. From 1992 to 1995, she served as project scientist of the Shuttle-Mir Program (STS-60, STS-63, STS-71, Mir 18, Mir 19), and as co-chair of the US–Russian mission science working group. From November 2003 to March 2005 she served as deputy chief of the astronaut office, then as chief of the astronaut office station operations branch until November 2005, when she began training as backup ISS Commander for Expedition 14. She served aboard two expeditions to the ISS. The first NASA science officer during Expedition 5 (June 5 to December 7, 2002), she conducted numerous experiments, in addition to installing the mobile case system, the S1 truss segment, the P1 truss segment, and micrometeoroid shielding on the Zvezda Service Module during an Orlan EVA. She commanded ISS Expedition 16 (October 10, 2007 to April 19, 2008), overseeing the first expansion of ISS living and working space in more than six years, including the Harmony connecting node, the European Space Agency's Columbus laboratory, the Japan Aerospace Exploration Agency's Kibo logistics pressurized module and the Canadian Space Agency's Dextre robot. Her six space walks are the most numerous of any woman.

Jeffrey N. Williams received a BS in applied science and engineering (1980) from the US Military Academy, an MS in aeronautical engineering (1987) and the degree of aeronautical engineer (1987) from the US Naval Postgraduate School, and an MA in national security and strategic studies (1996) from the US Naval War College. Commissioned as a second lieutenant upon graduation in 1980, he was designated an Army aviator in 1981 and subsequently served in the 3rd Armored Division's aviation battalion in Germany. In 1992, he was selected for the Naval Test Pilot School, and graduated first in his class before serving as an experimental test pilot and flight test division chief in the Army's airworthiness qualification test directorate at Edwards Air Force Base. During his Army assignment at NASA Johnson Space Center from 1987–1992, he served as a Shuttle launch and landing operations engineer and a pilot in the Shuttle avionics integration laboratory. In May 2000, he served as a the flight engineer and lead spacewalker for STS-101 (May 19–29, 2000), the third shuttle mission devoted to ISS construction. He was backup commander and Soyuz flight engineer for the 12th ISS Expedition in 2005. On ISS Expedition 13 (March 29 to September 28, 2006), he served as the flight engineer and science officer, during which he conducted two EVAs to resume ISS orbital laboratory assembly, oversaw the restoration of a three-person crew, and conducted numerous science experiments.

William V. Meyer received his BA in philosophy (1978) and BS in physics and mathematics (1983) from the University of Nebraska; his MS in physics, with a minor in engineering management (1987) from the University of Missouri-Rolla; and his PhD in physics (2002) from the University of Amsterdam. He has published extensively on dynamic and surface light scattering, and is a senior scientist at NASA Glenn Research Center, where he has worked since 1987. He has directed NASA advanced technology development projects in laser light scattering and surface light scattering, and is the deputy director at the National Center for Space Exploration Research, where his work as NASA project scientist for microgravity flight experiments probes how nature manifests order from disorder.

Ronald J. Sicker received his BSEE from Ohio University. From 1983-1989, he was a US Air Force flight test engineer on a fleet of EC-135 advanced range instrumentation aircraft, coordinating in-flight support for 38 space and missile tests, and managing the satellite and telemetry support for over 50 launches. Moving to NASA in 1989, he was an original member of the advanced communications technology satellite experiment office. From 1991 to 1993, he managed the design of the power protection system for the Space Station (Freedom) which became the ISS design. From 1994 to 2004, he managed the space acceleration measurement system and the orbital acceleration research experiment payloads for 15 Shuttle, Shuttle/Mir and ISS launches. From 2004 to 2005, he served as deputy project manager of the satellite based technology/advanced communication navigation and surveillance architectures system technology project. In 2006, he developed architecture requirements for the new ARES and ARES-5 launch vehicles for the Level II

Constellation program. Since August 2007, he has served as project manager for the ISS fluid integrated rack and light microscopy module, launched August 25, 2009, and is managing their outfitting and on-orbit operations.

Mark Christiansen received his BA, Phi Beta Kappa, in English (1990) from Pomona College, and began his career at LucasArts Entertainment. He is the author of *After Effects Studio Techniques* (Adobe Press). He has supervised visual effects for features films such as *All About Evil* and created visual effects and animations for features including *Avatar*, *Pirates of the Caribbean 3*, *The Day After Tomorrow*, and *Spy Kids 3D*. Clients of his company, Flowseeker LLC, include Sony, Adobe, Cisco, Sun, Cadence, Seagate, Intel and Medtronic, and his broadcast motion graphics work has appeared on HBO and the History Channel. Mark's roles have included producing, directing, designing and on-set and post production supervision. Mark is a founding author at ProVideoCoalition, an instructor at fxPhd.com and Academy of Art University, and a guest host on podcasts including the VFX Show (Pixel Corps).

Andrew B. Schofield received his PhD (1994) from the school of chemistry at the University of Bristol. He currently works in the school of physics at Edinburgh University. His research interests include the preparation of novel colloidal particles and their applications.

David A. Weitz received his BSc with honors in physics (1973) from the University of Waterloo, and AM (1975) and PhD (1978) in physics from Harvard University. From 1978 to 1995, he worked at Exxon Research and Engineering as a group leader in interfaces and inhomogeneous materials (1987–1989) and the science area leader in complex fluids (1989–1993). From 1995 to 1999, he was a Professor of Physics at the University of Pennsylvania, before becoming a Gordon McKay Professor of Physics and of Applied Physics at Harvard University (1999–2005). In 2005, he was appointed Mallinckrodt Professor of Physics and of Applied Physics at Harvard, and also currently serves as a professor of systems biology, director of the Harvard Materials Research Science and Engineering Center, co-director of the Kavli Institute for Bionano Science and Technology, and co-director of the BASF Advanced Research Initiative at Harvard. His research group focuses on the physics of soft condensed matter, colloidal dispersions, foams and emulsions, and biomaterials; the mechanics of biomaterials and cell rheology; microfluidic techniques for new complex fluid structures, bio-chemical assays, screening; synthesis of new soft materials; engineering structures for encapsulation; new optical measurement techniques for dynamics and mechanics of random systems; and multiple scattering of classical waves.