18th World IMACS / MODSIM Congress, Cairns, Australia 13-17 July 2009
http://mssanz.org.au/modsim09

Two and Three-Dimensional Image Deconvolution on Graphics Hardware

Luke Domanski, Pascal Vallotton and Dadong Wang

CSIRO Mathematical and Information Sciences, Biotech Imaging
Email: luke.domanski@csiro.au

Abstract: The process of image formation in an optical microscope or similar imaging device results in a distortion of the true object image due to diffraction effects and out-of-focus blurring. This distortion can greatly limit the resolution of the imaging device, particularly in the case of 3D microscopy where the axial resolution is impeded by the contribution of out-of-focus signal from an extended area of the object outside a recorded focal plane.

Fortunately the image formation process can be modelled as a convolution of the original object with the impulse response of the device, that is, the image of a point energy source. As such, an approximation of the original object image can be derived through an inversion of this model – a deconvolution. Unfortunately the inclusion of random noise in the image recording process makes direct methods of inverting this model less than ideal as they are prone to amplification of noise. A class of popular approaches is to use iterative methods which try to account for the noise and progress towards an acceptable approximation of the real image. These methods are often computationally intensive, requiring a reapplication of the image formation model to successive estimates of the real image. In addition, biological and medical research organisations are collecting 3D image data on an increasingly large scale, and there is a demand to process this data in a timely manner.

In this paper we investigate the use of graphics processing units (GPUs) to accelerate the execution of one such iterative algorithm, the Richardson-Lucy (RL) algorithm. Modern GPUs are highly parallel commodity processors containing hundreds of cores and capable of executing thousands of threads concurrently. GPUs can be utilised to accelerate a wide variety of compute intensive algorithms. As their programmability has improved over the past decade, significant effort has been invested in performing general purpose computing on GPUs (GPGPU). Until recently GPGPU algorithms had to be implemented using a combination of graphics application programming interfaces (APIs, e.g. OpenGL, DirectX) and shader languages, which impose a graphics focused conceptual view of the underlying hardware and hide a number of important hardware capabilities from the programmer. The advent of GPGPU programming languages such as CUDA, Brook, and OpenCL have made a number of these capabilities more accessible, paving the way for more efficient algorithms, and have seen the use GPGPU approaches in new application areas.

We compare performance results for a number of 3D Richardson-Lucy implementations on both the CPU and GPU, showing that our best GPU implementation, using Fourier space convolutions, significantly outperforms our best CPU implementation, which uses a publicly available and highly optimised Fast Fourier Transform (FFT) library.

```
Richardson-Lucy(O, PSF$^F$, N) return E {
E = O
for N iterations
        /* apply imaging model to estimate */
        $E^F$ = gpu_fft(E)
        $B^F$ = gpu_multiply($E^F$, $PSF^F$)
        B = gpu_ifft($B^F$)

        /* captured image divided by blurred estimate */
        R = gpu_divide(O, B)

        /* calculate correction vector */
        $R^F$ = gpu_fft(R)
        $C^F$ = gpu_multiply($R^F$, $PSF^{*F}$)
        C = gpu_ifft($C^F$)

        /* apply correction vector */
        E = gpu_multiply(E, C)
end
}
```

Figure 1: Frequency based GPU Richardson-Lucy algorithm. Variables O, E, B, R, C and PSF = original image, current estimate, blurred estimate, ratio, correction and psf respectively (stored in GPU memory). * denotes complex conjugate.

Keywords: *graphics processing units (GPU), parallel processing, image restoration, deconvolution, microscopy, Fast Fourier Transform*

1. INTRODUCTION

Blurring effects inherent to the process of image formation limit both the contrast and resolution of a microscope, restricting the accuracy and scale at which we can quantify and examine microscopic structures. In 3D light microscopy the blurring is predominately caused by diffraction and the contribution of out-of-focus light from regions of the object out of the focal plane. Optical aberrations, intensified by using the microscope outside of its design conditions (Gibson & Lanni, 1991), further degrade image quality.

The blurring caused during image formation can be characterised by the impulse response of the imaging device. The impulse response is the image of a point source object captured by the imaging device and is often referred to as the point spread function (PSF). With this information a linear model of the imaging system can be defined as a convolution of the true object *f(x)* with the PSF *p(x,ξ)* as follows.

$$g(x) = \int f(\xi)p(x,\xi)d\xi \qquad (1)$$

When $p(x,\xi)=p_i(x-\xi)$ the PSF is assumed to be spatially invariant (SI). In other words, the image of a point source object is identical regardless of where it is positioned in the sample. This is often not the case, particularly for 3D light microscopy (Gibson & Lanni, 1991), and the PSF may change based on axial and lateral location. In these cases a spatially variant (SV) description of the PSF provides a more accurate model of the imaging system, and *p* becomes a function of spatial location $p(x,\xi)=p_v(\xi,x-\xi)$ giving the intensity of light at image point *x* produced by a point light source at point *ξ* in object space (Preza & Conchello, 2003). Given these models of image formation the process of finding the unknown true object *f(x)* from the observed or captured image *g(x)* becomes one of inverting equation 1, that is, deconvolving the captured image using a suitable expression of the PSF.

It is well known from the Convolution Theorem for Fourier transforms that a convolution in the spatial domain is equivalent to an element-wise multiplication in the Fourier domain. Hence, for the spatially invariant model, equation 1 becomes

$$G(\omega) = F(\omega)P(\omega)\,, \qquad (2)$$

where capital letters denote the Fourier transform of the associated lower case functions. The desired deconvolution can then be performed using a straight-forward division in the Fourier domain.

$$F(\omega) = G(\omega)/P(\omega)\,. \qquad (3)$$

The true image is obtained by performing an inverse Fourier transform on the result. Handling the SV model is not as straight-forward, and will be discussed later.

When dealing with discrete images the integral operators above are replaced by the appropriate summation operators, and the Fourier transforms can be performed using the Discrete Fourier Transform (DFT) with the PSF padded to the size of the real image. Fast algorithms known collectively as Fast Fourier Transforms are available for computing the DFT, and provide an efficient means of performing both convolution and deconvolution using a computer. Such an approach to inverting the imaging model rarely performs well on real images though. The process of capturing an image using a digital camera introduces random noise *n* due to statically fluctuations in the measurement device. In practice the imaging model therefore becomes

$$g(x) = \int f(\xi)p(x,\xi)d\xi + n(x)\,. \qquad (4)$$

Because the noise is random and both the noise and true image are unknown, there is no certain method of separating the noise contribution in the final image from the contribution of the true image. Applying a deconvolution as described above in this situation results in an amplification of the noise (Lucy, 1994) which can render the output “deprived of any physical meaning” (Bertero & Boccacci, 2002).

A common approach for addressing the inversion of equation 4 is to use iterative algorithms that account for the statistical properties of noise, or regularise its effects in some way, while converging towards an optimal solution. A popular algorithm for achieving this is the Richardson-Lucy algorithm (Richardson, 1972; Lucy, 1974) which provides the Maximum Likelihood estimator for *f(x)* when the noise is modeled using Poisson statistics.

The algorithm is derived from an expression of the inversion problem using Bayes’ Theorem as described in (Richardson, 1972) and (Lucy, 1974), and is defined in the discrete case by

$$f_{r+1}(\xi) = f_r(\xi) \sum_x \frac{g(x)}{g_r(x)} p(x,\xi) \text{, where} \qquad (5)$$

$$g_r(x) = \sum_\xi f_r(\xi) p(x,\xi) \qquad (6)$$

It starts with f_0 as an estimate of f and proceeds to re-evaluate the estimate as shown for a given number of iterations or until some stopping criterion is reached. Notice that the algorithm does not actually invert the imaging model directly, instead, it: a) applies the imaging model to the estimate f_r producing a blurred estimate g_r, b) creates a correction factor by convolving the ratio of the observed image g to blurred estimate g_r by the "transpose" of the PSF (note the domain of summation), d) multiplies the current estimate f_r by the correction to get a new estimate.

Unfortunately this algorithm can become computationally intensive when a large number of iterations are required. From equations 5 and 6 we can see that each iteration of the algorithm requires the calculation of two convolutions plus a complete element-wise division and multiplication of the image. In this paper we investigate the use of graphics processing units (GPUs) to accelerate the Richardson-Lucy algorithm.

2. BACKGROUND AND RELATED WORK

2.1. Graphics Processing Unit

We will focus on CUDA and NVIDIA GPUs in our discussions (NVIDIA, 2008). A GPU consists of a number of multiprocessors (MPs) each with a set of 32 bit registers, private on-chip parallel shared memory, and managed read-only constant and texture memory caches. Each multiprocessor contains eight scalar processor cores (SPs) that share the register and memory resources of their MP, and execute the same instruction simultaneously at each instruction cycle. The MP schedules, switches, and executes threads in fixed groups of 32 threads called a warp. The same instruction is executed for all threads of the warp before proceeding to the next instruction or executing a different warp. Threads have general read-write access to device global memory attached to the graphics board, but this is not cached and incurs significant overhead compared with accessing the on-chip shared memory. Threads are indexed over a problem domain using a hierarchy of grids and blocks. A block is an N-Dimensional (N=1..3) array of threads and a grid is a KD (K=1,2) array of blocks. A block is assigned to a single MP for its lifetime and its threads can synchronise their execution and share data via MP shared memory. The number of blocks that can run on an MP at one time is limited by the register and shared memory resources required per block.

Significant performance benefits exist when a half-warp (first or second 16 threads of warp) accesses both global and shared memory in a particular pattern. When the 16 threads access consecutive elements of global memory and the first thread's access is aligned to particular segments of memory, the accesses are coalesced into a single memory transfer instead of 16 serialised transfers. Coalesced accesses on newer GPU hardware are slightly more relaxed, however, optimal results are still achieved using the rules above. Shared memory allows parallel accesses by threads of a half-warp that access different shared memory banks. Memory is partitioned across 16 banks in 32bit words such that the a^{th} word in memory is assigned to bank $b=a \bmod 16$. Accesses are serialised on the order of the minimal overlapping set of bank accesses (NVIDIA, 2008).

2.2. Convolutions and Discrete Fourier Transforms on the GPU

Spatial convolution is a fairly straight-forward task to perform on graphics hardware. Some graphics hardware provides operations for 2D convolution by small filters and Hopf and Ertl (1999) have shown how to apply these to the construction of 3D convolutions. A number of convolution approaches have been implemented using graphics APIs and shaders (Bjorke, 2004; James & O'Rorke, 2004; Viola, 2002; Hadwiger et al., 2001). A common approach is based on shifting and accumulating an image into a buffer using a single kernel weighting for each image, while others directly gather image and kernel values within a pixel shader program using texture accesses and produce the pixel output value by a standard multiply-summation of the values. Since the advent of CUDA, efficient algorithms for 2D convolution have been described (Podlozhnyuk, 2007), however, they gain much of their performance by using separable filters. We make no assumptions on separability and hence use a conventional gather-multiply-sum approach on a per pixel basis within the CUDA kernel.

FFTs have been implemented on the GPU using graphics APIs and shaders (Moreland and Angel et al., 2003; Govindaraju et al., 2006; Spitzer, 2003), however, most results were only comparable with the performance of optimised CPU FFT libraries such as FFTW and MKL. With the release of CUDA came NVidia's CUFFT 1.1 library which significantly outperformed previous GPU based FFT implementations as well as optimised CPU libraries. Since then marked improvements have been made in the performance of FFT algorithms on the GPU (Govindaraju et al., 2008; Volkov and Kazian, 2008; Nukada et al. 2008). In this work we utilise GPU FFT algorithms as "black-box" libraries and will not concern ourselves with their internal details.

2.3. Parallel Richardson-Lucy

A common approach to implementing the Richardson-Lucy algorithm in parallel on a cluster of PCs (Boden et al., 1996; Shearer et al., 2001) is based on the sectioned method of Trussell and Hunt (1978a). In such an approach the image can be segmented into a number of abutting sub-tiles and the RL algorithm is performed on each tile individually. A guard-band of half the PSF diameter is included around each tile to accommodate for the additional information required during convolution at tile boundaries. On completion the guard-bands are discarded and the tiles are combined to form the final deconvolved image.

Since the processing of each tile is performed in isolation, the tiles can easily be distributed to multiple PCs or processors and processed in parallel. The sectioned method also provides a convenient way of incorporating a SV model in a piece-wise manner (Trussell and Hunt, 1978b). By processing each tile using a spatially invariant model, but using a different PSF for each tile base its spatial location, spatially variability is achieved at the tile level. Tile size can be adjusted to accommodate greater or less extents of variability, thus minimising deconvolution artefacts between adjacent tiles. We present one such approach for the GPU.

An alternate or complementary approach is to parallelise the FFT algorithm used to facilitate frequency domain convolutions. Extensive effort has been applied to implementing GPU accelerated FFTs as discussed above. This type of acceleration could be used for both sectioned methods (FFT on individual tiles) and non-sectioned methods (FFT on whole image). Fung & Mann (2008) provide brief results for a non-sectioned GPU accelerated RL algorithm using CUFFT and custom kernels. They show a 9.8-21x speedup over the RL implementation in Matlab for a single test case, but do not provide discussions of the algorithm. We discuss such an implementation in more detail.

3. IMPLEMENTATION ON THE GPU

We examine two implementations of the Richardson-Lucy algorithm on the GPU, one that incorporates frequency domain convolutions and one that uses spatial domain convolutions.

3.1. FFT based approach

For the frequency domain approach we use a publically available FFT library for the GPU to perform the Fourier Transforms and implement other parts of the algorithm using customised GPU kernels. The main iterative control loop of the algorithm is implemented on the CPU and the loop body consists entirely of calls to the FFT library and our GPU kernels (figure 1). A major limiting factor in GPU computing is the transfer of data from CPU host memory to GPU device memory via the PCI Express bus. Implementing all algorithm operations on the GPU rather than having to read values back and forth to the CPU between each convolution not only provides increased parallelism over the CPU for performing these operations, but allows us to keep host-to-device transfers at a minimum.

The two convolutions required in the RL iteration are performed using a combination of FFT library calls and a customised kernel for the element-wise multiplication. While the input image data is stored as real valued numbers, the FFT will perform a real-to-complex transform, which results in complex output. The kernel for element-wise multiplication must therefore perform complex multiplication. Since the image array and the padded PSF array are of the same size, and share the same storage pattern in memory (e.g. row-major), the operation can be performed on the data as a 1D vector, regardless of the image dimensions. Such an ordering of values facilitates a simpler construction of the thread block/grid, and more efficient memory access patterns (§2.1). The other major operations, i.e. the original image to blurred estimate ratio and the application of correction vector to the estimate, are performed element-wise on real values using custom written division and multiplication kernels respectively.

Almost all input and output variables of the FFT and custom kernel functions are of the order of the original image size, and must be stored in GPU device memory upon function invocation. Modern high-end GPUs have between 756MB and 4GB of device memory, while a typical 1024x1024x16 real valued image consumes 64MB (1MB = 1024 kB). Storing multiple variables of this size can be costly in memory usage. To conserve space, operations will be performed in-place where appropriate, so that the same piece of memory can be used to store both the input and output. The most suitable variables for in-place calculation or memory reuse are those whose values will not be required after they have been used as input to an operation. The RL algorithm requires up to four variables to remain persistent within and between algorithm iterations: the original image (O in figure 1), the current estimate (E), and the Fourier transforms of the PSF and its complex conjugate (PSF^F and PSF^{*F}). All other variables are candidates for reuse. Examining figure 1 we see that none of these remaining variables are used more than once after they are calculated, nor are any of them used simultaneously as input to a GPU function. Given that both FFTs and element-wise operations can be performed in place, we require only one segment of memory to represent all of these variables. In practice we use two memory segments, a and b, to avoid unnecessary padding as discussed bellow. Variables share these segments as such: E^F -> B^F -> B -> R -> R^F -> C^F -> &b, and C -> &a.

Although a DFT performed on $N=N_1xN_2x\ldots xN_d$ real value samples (N_k = size of kth dimension) produces N complex values, almost half of these samples occur redundantly as the conjugate of other samples. A common practice is to discard the redundant coefficients and store only $N_1xN_2x\ldots x(N_d/2+1)$ complex values. Since complex values require twice the space of real values, the memory required to store the complex FFT output will be slightly larger than that required to store the real valued image. When performing in-place FFT calculations one must allow for this extra space by padding the real valued image along the last dimension within the larger array. The length of an array dimension including padding is known as pitch. For the FFT library we utilise, the padding occurs at the end of rows. In a row-major storage scheme this raises difficulties when performing element-wise operations between real valued images stored in a standard manner and ones of the same "size" stored using padding. This is because associated image samples are not at the same memory offsets from their array base addresses. To handle this situation one can either pad all real valued images so they share a common storage arrangement, or create GPU kernels that handle pitched access. We have implemented kernels that account for pitch to avoid unnecessary padding, however, we may experiment with padding in the future as it could provide more efficient access patterns in the kernel (§2.1).

3.2. Spatial domain approach

For the spatial domain convolution approach we use a single kernel that implements the entire RL algorithm on the GPU before returning. The implementation is based on the conventional sectioned method for parallelisation of the RL algorithm, where we map the computations for a given tile to a single multiprocessor of the GPU. All threads processing a tile can then take advantage of on-chip shared memory and synchronisation barriers to cooperatively process the tile without writing back to global device memory.

As discussed in section 2.3, the RL algorithm can be performed by breaking the image into small tiles (with guard-bands) and running the algorithm individually on each. Parallelism can be achieved easily by processing multiple tiles simultaneously. This approach works particularly well on distributed memory clusters, as a single tile can be assigned to each processor and no communication is required between the processors (tiles). The properties that make this approach attractive for a distributed memory cluster also make it attractive for GPU implementation. As previously discussed (§2.1), threads that run on a single GPU MP can synchronise their execution and communicate values to one another via this shared memory. The shared memory of one multiprocessor can not be accessed by other multiprocessors, nor are there any convenient mechanisms to synchronise threads on different multiprocessors. Communication and synchronisation between multiprocessors is therefore inconvenient and costly, requiring data writes to global device memory and termination of a kernel to achieve global synchronisation. In addition, the shared memory assigned to a thread is unlikely to remain persistent across a kernel invocation. Like a single processor of a cluster, a single GPU multiprocessor can therefore process a tile quickly and efficiently using its own local resources, while communication with other processors is far more costly.

Each tile of the image will therefore be allocated to a single thread block that will run on a multiprocessor. The block of threads will be indexed over the tile's local spatial domain as a 2D or 3D box, and each thread will handle the processing of the associated tile pixel. The two ND convolutions in the kernel will be performed using standard N-level nested for loops with a multiply-add in the inner loop. All other calculations can be performed using a few arithmetic operations.

During the convolutions to obtain the blurred estimate and correction vector the threads will require values of the current estimate and ratio that were calculated by other threads in the block. As such, a barrier synchronisation should be called prior to each convolution to avoid read-before-write access problems. These commonly read and written values, along with the original image, will be stored in shared memory to facilitate efficient access. Reading and writing these values to global memory over a large number of RL iterations would otherwise incur a significant performance penalty. The commonly accessed PSF data remains constant throughout the algorithm and can be placed in read-only constant memory. Constant memory constitutes part of device global memory, however, constant memory accesses are automatically cached on the multiprocessor in a 16KB cache. When all threads of a warp access the same cached element of constant memory it is as fast as accessing a register (NVIDIA, 2008). This will be the case for PSF accesses within the convolutions as they are dictated directly by the parameters of the *for loop*, and the loops will be performed in lock step by the threads of a warp. Values calculated for other array variables, such as the blurred estimate and correction vector, remain local to each thread and can be stored in thread registers. This represents the data arrays implicitly across registers and provides a performance advantage compared to memory access. It also avoids the need to consider in-place operations and variable reuse as the CUDA compiler will reuse register resources appropriately.

Performance is not the only issue dictating our selection of variable storage above. Recall that the RL algorithm requires a number of image-sized variables to be maintained at any one time. The shared memory resident on a multiprocessor is 16kB shared between all blocks currently occupying the multiprocessor. For each tile "image" in the sectioned method we require both the tile itself and a guard band half the width of the PSF around the entire tile. Taking a 16x16 pixel tile and a 17x17 PSF results in approximately 4kB for real valued data, meaning we can store a maximum of 3 or 4 tiles in shared memory. In our implementation the shared memory requirement can be restricted to one tile image per block using the storage options above, because the same space can be used to store the blurred estimate and ratio without conflict. When considering 3D images, however, the shared memory resource soon become inadequate. For an 8x8x8 tile and 9x9x9 PSF we hit the 16kB barrier for a tile and the kernel invocation begins to fail. Point spread functions more than twice this size are not uncommon in 3D widefield microscopy. Even with smaller 3D tiles or PSFs it is unlikely that more than one block can share the multiprocessor at one time. This reduces the thread scheduler's ability to hide stalls and latencies by swapping in threads of another block. When PSFs reach a size that will exceed the shared memory resources using our implementation (31x31x1), we utilise kernels that use global memory only.

A maximum of 512 threads can be assigned to a block and it is generally recommended that each block contain somewhere between 128-256 threads (NVIDIA, 2008). We use a 16x16x1 block of threads for both 2D and 3D problems. To handle the much larger tile image each thread will perform the calculations for multiple pixels, by shifting the block of threads over the tile image domain. The threads will be shifted by the dimensions of the block, and for each location neighbouring threads will handle neighbouring pixels. This is more efficient than neighbouring threads making strided memory accesses and shifting the block by one pixel at a time due to the memory access versus performance benefits discussed earlier (§2.1).

4. RESULTS AND DISCUSSIONS

The algorithms described above were implemented in CUDA on an NVidia GTX 260 graphics card with 240 stream processors and 896MB of global memory (RAM). The host computer had a 2.49GHz Xeon Quad-Core processor running 32bit Windows XP, CUDA environment 2.1, CUFFT 2.1 for the GPU FFTs, and FFTW 3.1 (Frigo & Johnson, 2005) for the CPU FFTs. We compared our GPU algorithms against *non-sectioned* frequency and spatial domain implementations on the host. The CPU algorithms were single threaded except where FFTW used multi-threading. The code for the CPU algorithms where almost identical to their equivalent GPU algorithms, e.g. they were created by taking the GPU code and changing it slightly to run on the CPU and with FFTW. GPU versions include additional code to copy values from the host to the GPU, move values from device global memory to MP shared memory, and to iterate the 2D block of threads over the image tile instead of a single thread (CPU) in the spatial domain approach.

Figure 2 shows the results for 2D and 3D images. Although the sectioned algorithm maps well to the GPU processor architecture, the computational

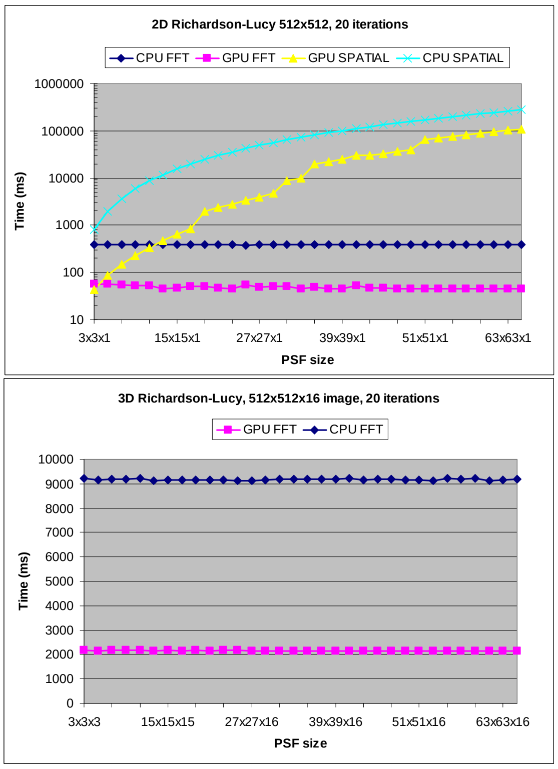

Figure 2: Results for 2D (top) and 3D (bottom) Richardson-Lucy on a 512x512(x16) floating point image using 20 iterations

complexity O(*p*x*n*x*m*x*o*) of a 2D spatial convolution is prohibitive (*p*=number of pixels, *n*x*m*x*o*=PSF size). We have excluded results for the 3D spatial domain approach due to its poor performance in 2D. Both the GPU and CPU non-sectioned frequency based approaches perform well, with the GPU version providing approximately 8.5x and 4.2x speedup for 2D and 3D images respectively. We have noted (§2.3) that an advantage of using sectioned methods is the straightforward integration of SV-PSF models. We also mentioned that frequency based deconvolution could be used in conjunction with a sectioned decomposition of the problem, and we will be experimenting with this in the future to take advantage of both methods.

5. CONCLUSION

We have presented and compared a number of parallel implementations of the Richardson-Lucy deconvolution algorithm on the GPU and CPU. We took two main approaches: one that applies frequency based convolution and utilises a CPU control loop with a combination of existing GPU FFT library calls and custom written kernels to parallelise individual algorithm operations, and one that applies spatial domain convolutions and performs the entire algorithm in a single GPU kernel. Results show that the non-sectioned FFT based algorithm is the fastest and outperforms a similar implementation using the accelerated CPU FFT library FFTW (4.2-8.5x speedup). The sectioned spatial domain approach is slower, but allows straightforward integration of spatially variant PSFs. Future work will involve implementing and testing a sectioned method that utilises frequency domain convolutions.

ACKNOWLEDGMENTS

This work was conducted under the Computation and Simulation Sciences Platform and Computational Imaging and Visualisation project within CSIRO. Thanks go to Yulia Arzhaeva and anonymous reviewers for reviewing this paper and providing useful comments.

REFERENCES

Bertero, M. & Boccacci, P. (2002), *Introduction to Inverse Problems in Imaging*, IOP Publishing.

Bjorke, K. (2004), High-Quality Filtering 'GPU Gems', Addison-Wesley Professional.

Boden, A. F.; Redding, D.; Hanisch, R. J. & Mo, J. (1996), 'Massively parallel spatially variant maximum-likelihood restoration of Hubble Space Telescope imagery', *J. Opt. Soc. Am. A* **13**(7), 1537-1545.

Frigo, M. & Johnson, S. G. (2005), 'The Design and Implementation of FFTW3', *Proceedings of the IEEE* **93**(2), 216-231.

Fung, J. & Mann, S. (2008), Using graphics devices in reverse: GPU-based Image Processing and Computer Vision, *in* 'Proceedings of the 2008 IEEE International Conference on Multimedia and Expo', pp. 9-12.

Gibson, S. F. & Lanni, F. (1991), 'Experimental test of an analytical model of aberration in an oil-immersion objective lens used in three-dimensional light microscopy', *J. Opt. Soc. Am. A* **8**(10), 1601-1613.

Govindaraju, N. K.; Larsen, S.; Gray, J. & Manocha, D. (2006), A memory model for scientific algorithms on graphics processors, *in* 'SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing', ACM, New York, NY, USA, pp. 89.

Govindaraju, N. K.; Lloyd, B.; Dotsenko, Y.; Smith, B. & Manferdelli, J. (2008), High performance discrete Fourier transforms on graphics processors, *in* 'SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing', pp. 1-12.

Hadwiger, M.; Theußl, T.; Hauser, H. & Gröller, E. (2001), Hardware-accelerated highquality reconstruction on PC hardware, *in* 'In Proceedings of the Vision Modeling and Visualization Conference 2001'.

Hopf, M. & Ertl, T. (1999), Accelerating 3D convolution using graphics hardware, *in* 'Visualization '99. Proceedings', pp. 471-564.

James, G. & O'Rorke, J. (2004), Real-Time Glow 'GPU Gems', Addison-Wesley Professional.

Lucy, L. B. (1974), 'An iterative technique for the rectification of observed distributions', *The Astronomical Journal* **79**(6), 745-754.

Lucy, L. B. (1994), 'Astronomical Inverse Problems', *Reviews in Modern Astronomy* **7**, 31-50.

Moreland, K. & Angel, E. (2003), The FFT on a GPU, *in* 'HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware', Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp. 112-119.

Nukada, A.; Ogata, Y.; Endo, T. & Matsuoka, S. (2008), Bandwidth intensive 3-D FFT kernel for GPUs using CUDA, *in* 'SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing', IEEE Press, Piscataway, NJ, USA, pp. 1-11.

Podlozhnyuk, V. (2007), Image Convolution with CUDA, 'NVIDIA CUDA SDK Whitepapers'.

Preza, C. & Conchello, J.-A. (2003), Image estimation accounting for point-spread function depth variation in three-dimensional fluorescence microscopy, *in* 'Three-Dimensional and Multidimensional Microscopy: Image Acquisition and Processing X, Proceedings of the SPIE', SPIE, , pp. 135-142.

Richardson, W. H. (1972), 'Bayesian-Based Iterative Method of Image Restoration', *J. Opt. Soc. Am.* **62**(1), 55.

Shearer, A.; Gorman, G.; O'Doherty, T.; van der Putten, W. J.; McCarthy, P. & Jelen, L. (2001), Parallel image restoration with spatially variant point spread function: description and first clinical results, *in* 'Medical Imaging 2001: Image Processing, Proceedings of SPIE', SPIE , pp. 787-795.

Spitzer, J. (2003), 'Implementing a GPU-Efficient FFT', SIGGRAPH '03 Course: Interactive Geometric & Scientific Computations with Graphics Hardware.

Trussell, H. & Hunt, B. (1978), 'Sectioned methods for image restoration', *Acoustics, Speech, and Signal Processing, IEEE Transactions on* **26**(2), 157-164. (a)

Trussell, H. & Hunt, B. (1978), 'Image restoration of space-variant blurs by sectioned methods', *Acoustics, Speech, and Signal Processing, IEEE Transactions on* **26**(6), 608-609. (b)

Viola, I. (2002), Applications of Hardware Accelerated Filtering, *in* 'Proceedings of the sixth Central European Seminar on Computer Graphics'.

Volkov, V. & Kazian, B. (2008), 'Fitting FFT onto thre G80 Architecture', UC Berkeley CS258 project report, http://www.cs.berkeley.edu/~kubitron/courses/cs258-S08/projects/reports/project6_report.pdf, retrieved on 01/21/2008.

NVIDIA (2008), 'NVIDIA CUDA Compute Unified Device Architecture Programming Guide', NVIDIA.