Technical Report

TR-2008-023

Image Processing on Modern CPUs and GPUs

by

Piotr Wendykier, James G. Nagy

Mathematics and Computer Science

EMORY UNIVERSITY

Image Processing on Modern CPUs and GPUs

PIOTR WENDYKIER and JAMES G. NAGY
Emory University[1]

Major breakthroughs in chip and software design have been observed for the last seven years. In October 2001, IBM released the world's first multicore processor - POWER4. Six years later, in February 2007, NVidia made a public release of CUDA SDK - a set of development tools to write algorithms for execution on graphic processing units (GPU). Although software vendors have started working on parallelizing their products, the vast majority of existing code is still sequential and cannot utilize modern multicore CPUs and manycore GPUs.

This article describes Parallel Colt - a multithreaded Java library for image processing. Besides its design and functionality, a comparison to MATLAB is presented. An ImageJ plugin for iterative image deblurring is described as a typical application of this library. Performance comparisons with MATLAB code (RestoreTools), including GPU computations via AccelerEyes' Jacket toolbox, are also given.

Categories and Subject Descriptors: D.3.2 [**Java**]: Scientific Computing; G.4 [**MATLAB**]: Image Processing

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: FFT, inverse problems, iterative methods, multithreading, regularization

1. MOTIVATION

For the last 40 years, the gain in CPU performance has been achieved by increasing the clock speed, execution optimization, and by increasing the size of on-chip cache. The 'clock race' ended in 2003, when all chip manufacturers reached hard physical limits: increasing heat generation, lack of suitable cooling hardware, increasing power consumption, current leakage problems, and increasing length of wire interconnects. October 2001 marks the beginning of a new era in CPU manufacturing when IBM released the world's first multicore processor - POWER4. Since then, all new processors have been designed to consist of two or more independent cores on a single die. Six years later, in February 2007, NVidia publicly released CUDA SDK [NVidia Corporation 2008], a set of development tools to write algorithms for execution on the graphic processing units (GPU). General-Purpose computation on GPUs (GPGPU) became available on virtually all desktop computers (AMD also has a toolbox for GPU computing). Although software vendors have started parallelizing their products, the vast majority of existing code (not only related to scientific computing) is still sequential. In practice, this means that only one-fourth of a quad-core CPU (currently a standard in a desktop PC) is utilized by a given program.

In this article we focus only on scientific applications, where parallel computing has been used since the 1960s (when the first supercomputers were developed). However, only recently have these programming paradigms become useful for software running on desktop and notebook computers. Here we demonstrate the advantage of exploiting modern computer architectures in scientific computing with multithreaded programming in Java and GPGPU in MATLAB. Furthermore, we show how these programming models can be

[1]Emory University, Dept. of Math and Computer Science, 400 Dowman Drive, W401, Atlanta, GA 30322. Research supported in part by the NSF under grand DMS-0811031.

applied to efficiently implement iterative algorithms to solve a class of ill-posed inverse problems in image processing. The result is a very efficient software package that can be used for real image processing applications on affordable desktop and laptop computers. In the following paragraphs we motivate our choice of these particular programming languages.

Although Java was not designed to be a scientific computing language [Byous 2003], it has several unique features that are attractive for high-performance scientific computing. First of all, Java is the most portable programming language ever implemented. Distributions are available for virtually all computing platforms. Second, since 2007, Java is an open source project so that anyone can modify and adapt it to their needs. Third, Java has native support for multithreading, and since version 5.0 [Sun Microsystems 2004] it is equipped with concurrency utilities in the `java.util.concurrent` package. Fourth, the performance of the latest version of Java (6.0) is comparable to the performance achieved by programs written in Fortran or C/C++[Amerdo et al. 2008]. Finally, sophisticated imaging functionality is built into Java, allowing for efficient visualization and animation of computational results. This is especially important for our work in image processing, but is also useful in many areas of scientific computation, such as computational fluid dynamics. However, because of certain design choices, there are also disadvantages of using Java in scientific computing. These include no primitive type for complex numbers, an inability to do operator overloading, and no support for IEEE extended precision floats. In addition, Java arrays were not designed for high-performance computing; a multi-dimensional array is an array of one-dimensional arrays, making it difficult to fully utilize cache memory. Moreover, Java arrays are not resizable, and only 32-bit array indexing is possible. Finally, GPGPU is not possible in Java. There exist libraries like JCufft [JavaGL 2008b] and JCublas [JavaGL 2008a] that provide Java bindings to CUDA, but they are only wrappers to underlying C code. To overcome these disadvantages, open source numerical libraries, such as Colt [Hoschek 2004] or JScience [Jean-Marie Dautelle 2007], have been developed. For our work, we are implementing a fully multithreaded version of Colt, which we call Parallel Colt [Wendykier 2008b].

In contrast to Java, MATLAB was initially developed only for scientific computations. MathWorks introduced multithreading in MATLAB R2007a, but even in the latest version (R2008b) the usage of multiple threads is very limited. In particular, most of the linear algebra algorithms, such as matrix decompositions, are still sequential. Similarly, there is no support for multithreaded Fast Fourier Transforms (FFT) [Cooley and Tukey 1965]. This situation will probably change in the next release, due to the fact that the Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [Buttari et al. 2007] is already available. We find the lack of threaded FFTs in MATLAB a little surprising, since these algorithms have long been supported in FFTW. GPU based computations are available in MATLAB through the third-party toolbox called Jacket [AccelerEyes 2008]. Besides the poor support for multithreading, the main disadvantage of using MATLAB in scientific computations, compared to our Java code, is the licensing (MATLAB is a commercial product) and source code availability.

The rest of the paper is organized as follows. In Section 2 we describe three numerical Java libraries: JScience [Jean-Marie Dautelle 2007], OR-Objects [DRA Systems 2000], MTJ [Bjørn-Ove Heimsund 2007], an image processing program called ImageJ [Rasband 2008] and a GPU computing toolbox for MATLAB – Jacket [AccelerEyes 2008]. Sec-

tion 3 presents a technical analysis of Parallel Colt [Wendykier 2008b] while in Section 4 we describe an application of our library to image deblurring, where we compare the performance of Java and MATLAB code. Finally, we summarize our work in Section 5.

2. RELATED WORK

Although many open source[2] Java packages for scientific computation exist, none are as comprehensive as MATLAB. Most of the Java libraries are targeted to solve problems from one particular field of study. We have chosen JScience, OR-Objects and MTJ as a software related to our work, since according to our knowledge these are the only freeware Java libraries with support for parallel computations.

JScience [Jean-Marie Dautelle 2007] is an open source package written by Jean-Marie Dautelle with the ultimate goal to "create synergy between all sciences (e.g. math, physics, sociology, biology, astronomy, economics, etc.) by integrating them into a single architecture". It supports multithreaded computations through Javolution (real-time programming library). Current features include modules for measures and units, geographic coordinates, mathematical structures (e.g. group, ring, vector space), linear algebra, symbolic computations, numbers of arbitrary precision, physical models (e.g. standard, relativistic, high-energy, etc.) and currency conversions. Nonetheless, JScience provides almost no support for image processing, its linear algebra module is very limited (only LU factorization), there is no class that represents a tensor (3D matrix), no matrix sub-ranging, and no FFTs.

Matrix Toolkits for Java (MTJ) [Bjørn-Ove Heimsund 2007] is a collection of matrices, linear solvers (direct and iterative), preconditioners, least squares methods and matrix decompositions written by Bjørn-Ove Heimsund. This library is based on BLAS [Blackford et al. 2002] and LAPACK [Anderson et al. 1999] for dense and structured sparse computations and on Templates [Barrett et al. 1994] for unstructured sparse computations. By default JLAPACK [Doolin et al. 1999] is used, but MTJ can be configured to use native BLAS and LAPACK libraries (such as ATLAS [Whaley and Dongarra 1998]). Moreover, the library supports distributed computing via an MPI-like interface. However, MTJ does not supply multithreading, tensors, complex matrices, matrix sub-ranging, and FFTs.

OR-Objects [DRA Systems 2000] is a collection of 500 Java classes developed by DRA Systems. It contains packages for linear programming, graph algorithms, matrix and linear algebra, numerical integration, probability and statistics, and geometry. Although OR-Objects is a freeware library, the source code is unavailable, which makes it much less attractive from our point of view. Analogous to JScience and MTJ, OR-Objects does not provide FFTs, tensors, complex matrices and its multithreaded functionality is limited only to BLAS.

ImageJ [Rasband 2008] is an open source image processing program written in Java by Wayne Rasband, at the U.S. National Institutes of Health (NIH). Besides having a large number of options for image editing applications, ImageJ is designed with pluggable architecture that allows developing custom plugins (over 300 user-written plugins are currently available). Due to this unique feature, ImageJ has become a very popular application among a large and knowledgeable worldwide user community. We use ImageJ in our work as a front-end for Parallel Colt.

[2]There are commercial Java numerical libraries, such as JMSL [Visual Numerics 2008], but we strongly believe that scientific software should be open source, so we do not compare our methods to these libraries.

Jacket [AccelerEyes 2008] is a commercial product being developed by AccelerEyes. It is a toolbox that enables a standard MATLAB code to be run on the GPU. Jacket does not introduce a new API, but instead it allows to write programs in the native M-Language, which are then automatically wrapped into a GPU compatible form. Currently Jacket supports only NVidia graphic cards and, compared to MATLAB, its functionality is very limited. In particular, none of the LAPACK routines are supplied. Despite this limitation, it already allows to speed up many image processing algorithms by supporting fast Fourier transforms on GPUs. In Section 4 we show how image deblurring algorithms can benefit from using Jacket.

3. PARALLEL COLT

3.1 Colt

Colt [Hoschek 2004] is an open source library for high-performance scientific computing in Java written by Wolfgang Hoschek at CERN. It features an efficient and usable data structure and algorithms for data analysis, linear algebra, multi-dimensional arrays, statistics, histogramming, Monte Carlo simulation and concurrent programming. The project is currently inactive, the latest version (1.2.0) was released in September 2004. We have chosen to adapt Colt to fit our purpose of having a powerful computing engine for image processing. Our choice was motivated primarily by the fact that Colt has support for uniform, versatile and efficient multi-dimensional arrays (matrices)[Hoschek 2000]. In particular, *views* operations defined on multi-dimensional arrays allow sub-ranging, striding, transposition, slicing, index flipping, cell selection as well as sorting, permuting and partitioning of the elements. This is almost the same range of functionality as provided by MATLAB. In the rest of this section we summarize all the changes and new functionalities that we introduced in Parallel Colt.

3.2 Concurrency

Multithreading in Colt 1.2.0 is limited to a subset of BLAS routines: matrix-matrix and matrix-vector multiplications as well as the generalized matrix scaling/transform. All other algorithms included in the library are sequential. Moreover, Colt uses Doug Lea's `EDU.oswego.cs.dl.util.concurrent` package for concurrency instead of improved, more efficient and standardized classes (`java.util.concurrent`) which are included in a standard Java distribution since version 5.0. Concurrency in Colt requires setting a maximum number of threads before the first use of SMP BLAS, as opposed to Parallel Colt, where multithreading is enabled by default (if the number of available CPUs is greater than one). Java utility classes for concurrent programming contain the cached thread pool feature that we have found to be very useful. This type of pool creates new threads as needed, and reuses previously constructed threads when they become available, thereby improving the performance of programs that execute many short-lived asynchronous tasks. It turns out that almost all element-by-element operations and BLAS routines can be split into asynchronous tasks. Therefore, Parallel Colt uses the cached thread pool for low-level concurrency.

3.3 Multidimensional Arrays

There are many problems in image processing where double precision is unnecessary. This is usually the case when the source image is saved in a grayscale 8-bit format (integers from

0 to 255). Form the computational point of view, single precision has two advantages over double precision: arithmetic operations are faster with single precision numbers and they require only half the storage of double precision numbers. All algorithms in Colt 1.2.0 that use floating-point numbers are implemented in double precision, in particular, only double precision multidimensional arrays are available. Therefore, in Parallel Colt we have added single precision equivalents to all double precision based objects.

Another new and important type of object added to Parallel Colt is a multidimensional array of complex numbers. This object is essential for operations involving Fast Fourier Transforms. Due to the lack of primitive type for complex numbers in Java we decided to store them in a one-dimensional array of doubles (or floats) in the interleaved fashion (the real and imaginary parts). This type of storage guarantees much better performance than defining a new object that represents a complex number, and then storing an array of such objects. Currently Parallel Colt does not support linear algebra algorithms (except matrix-matrix and matrix-vector multiplications) for complex matrices.

Colt is equipped with three different sorting algorithms: quicksort, mergesort and binary search, which complement the `java.util.Arrays` class. Moreover, these algorithms are used to sort elements of multidimensional arrays. In Parallel Colt we have implemented a multithreaded version of quicksort that works both on arrays of primitive types and arrays of objects.

Finally, Parallel Colt's implementation of multidimensional arrays contains many additional methods: getMaxLocation, getMinLocation, getNegativeValues, getPositiveValues, normalize, vectorize, reshape, fft, ifft, fft2, ifft2, fft3, ifft3, fftColumns, ifftColumns, fftRows, ifftRows, fft2Slices, ifft2Slices, getFft, getIfft, getFft2, getIfft2, getFft3, getIfft3, getFftColumns, getIfftColumns, getFftRows, getIfftRows, getFft2Slices, getIfft2Slices, dht, idht, dht2, idht2, dht3, idht3, dhtColumns, idhtColumns, dhtRows, idhtRows, dht2Slices, idht2Slices, dct, idct, dct2, idct2, dct3, idct3, dctColumns, idctColumns, dctRows, idctRows, dct2Slices, idct2Slices, dst, idst, dst2, idst2, dst3, idst3, dstColumns, idstColumns, dstRows, idstRows, dst2Slices, idst2Slices.

3.4 Linear Algebra

Most of the linear algebra functionality, including all matrix factorizations, is still sequential in Parallel Colt. There are two reasons why we have not parallelized this part of the library yet. First of all, it is an extremely difficult task. Second, it is not a crucial functionality for image processing applications. Only recently the first version of LAPACK for multicore platforms, called PLASMA [Buttari et al. 2007], has been released. Currently PLASMA only supports LU, Cholesky and QR matrix factorizations. The most important matrix factorization for image processing applications is the Singular Value Decomposition (SVD), which we will try to port to Java as soon as it is available in PLASMA. Parallel Colt currently implements two SVD algorithms - one is the original Colt version, which is essentially a slightly modified algorithm from Jama [Hicklin et al. 2005], and the other is a divide-and-conquer routine from JLAPACK (`dgesdd`). All classes of JLAPACK are available in Parallel Colt, but currently only wrappers for SVD are provided. Note that our current use of SVD in image processing is within a Krylov subspace method that enforces regularization on a (small) projected linear system; see [Chung et al. 2008].

Besides including JLAPACK in Parallel Colt, we have also added the following linear algebra algorithms: Kronecker product of 1D matrices (complex and real), Euclidean norm of 2D and 3D matrices computed as a norm of a vector obtained by stacking the columns of

the matrix on top of one another, Frobenius norm of 2D complex matrices, and backward and forward substitution algorithms for 2D real matrices.

3.5 Trigonometric Transforms

There are four trigonometric transforms available in Parallel Colt: Discrete Fourier Transform (FFT) [Cooley and Tukey 1965], Discrete Hartley Transform (DHT) [Hartley 1942], Discrete Cosine Transform (DCT) [Ahmed et al. 1974] and Discrete Sine Transform (DST) [Yip and Rao 1980]. All of these transforms are implemented as public methods for 1, 2, and 3-dimensional dense matrices (see Section 3.3). In addition, they can be applied to matrix subranges. To provide this functionality in Parallel Colt, we have integrated our library called JTransforms.

JTransforms [Wendykier 2008a] is the first, open source, multithreaded FFT library written in pure Java. The code was derived from the General Purpose FFT Package by Ooura [Ooura 2006] and from Java FFTPack [Baoshe Zhang 2005] by Zhang. Ooura's library is a multithreaded implementation of the split-radix FFT algorithm in C and Fortran. In order to provide more portability both Pthreads and Windows threads are used in the implementation. Moreover, the code is highly optimized and in some cases runs faster than FFTW [Frigo and Johnson 2005]. Even so, the package has several limitations arising from the split-radix algorithm. First of all, the size of the input data has to be a power-of-two integer. Second, the number of computational threads must also be a power-of-two. Finally, one-dimensional transforms can only use two or four threads. To overcome the power-of-two limitation we have adapted Zhang's Java code which is a straightforward translation of the mixed-radix algorithm from FFTPACK [P. N. Swarztrauber 2004]. Since Java FFTPack contains only sequential algorithms for 1D transforms (real and complex), we have implemented multithreaded 2D and 3D transforms. As a result, the current version of JTransforms works for arbitrarily sized data.

There are some important distinctions between our Java code and Ooura's C implementation. First, JTransforms uses a thread pool, while the original package does not. Although thread pooling in Pthreads is possible, there is no code for this mechanism available in the standard library, and therefore many multithreaded applications written in C do not use thread pools. This has the added problem of causing overhead costs of creating and destroying threads every time they are used. Another difference between our code and Ooura's FFT is the use of "automatic" multithreading. In JTransforms (and in Parallel Colt), threads are used automatically when computations are done on a machine with multiple CPUs. Conversely, both Ooura's FFT and FFTW require manually setting up the maximum number of computational threads. Lastly, JTransforms' API is much simpler than Ooura's FFT, or even FFTW, since it is only necessary to specify the size of the input data; work arrays are allocated automatically and there is no planning phase.

3.6 Accuracy

There are two aspects about the accuracy of floating-point arithmetic in Java. The first is related to the internal design and implementation of Java's floating-point arithmetic. There are several flaws in this implementation.[W. Kahan and J. D. Darcy 1998]. First of all, Java does not completely conform to IEEE 754 standard, since it does not support the flags for IEEE 754 exceptions: Invalid Operation, Overflow, Division-by-Zero, Underflow, Inexact Result. In other words, no event occurs when the value of a floating-point number becomes either Infinity or NaN. Moreover, Java does not provide the IEEE extended preci-

sion in spite of the fact that over 95% of today's computers have hardware that can support these types of numbers. Finally, of two traditional policies for mixed precision evaluation, Java chose the worse. However, our experience shows that Java's floating-point arithmetic is good enough for applications in image processing. This is supported by the fact that usually the pixels of an image are stored as integers (byte and short) or as a single-precision floats, thus the double (or even single) precision arithmetic provides a sufficient amount of accuracy.

Another aspect of the accuracy is related to the stability of an algorithm and round-off errors. In the previous release of Parallel Colt we observed inaccurate results for trigonometric transforms when the size of the input data was a number that contains a large prime factor. The problem resulted when the mixed-radix FFT algorithm that was used for this case. To compute the FFT (or any other transform available in Parallel Colt) of a large prime factor, a slow discrete Fourier transform algorithm ($\mathcal{O}(n^2)$) was used. It is known [Schatzman 1996], however, that the root mean square error in this case is $\mathcal{O}(\sqrt{n})$, where n is the size the input data. The original FFTPACK library is also burdened with this error. In the current version of Parallel Colt (and JTransforms) we have fixed all the accuracy issues by implementing Bluestein's FFT algorithm [Bluestein 1968]. Figures 1 and 2 show that the FFTs in Parallel Colt are as accurate as FFTs in MATLAB. Jacket's FFTs, on the other hand, are much less accurate, especially for prime numbers.

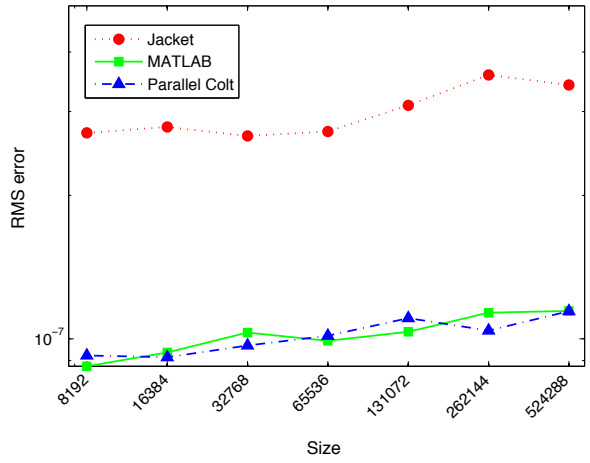

Fig. 1. Accuracy of complex, single precision, 1D FFT (power of two sizes). The horizontal axis is the root mean square error, $\frac{\|\mathbf{x} - \text{ifft}(\text{fft}(\mathbf{x}))\|_2}{\|\mathbf{x}\|_2}$, where $\mathbf{x}$ is a vector whose size is shown on the vertical axis.

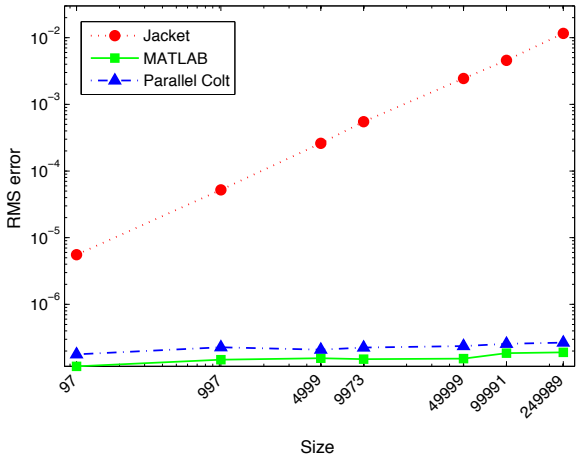

Fig. 2. Accuracy of complex, single precision, 1D FFT (prime sizes). The horizontal axis is the root mean square error, $\frac{\|\mathbf{x} - \text{ifft}(\text{fft}(\mathbf{x}))\|_2}{\|\mathbf{x}\|_2}$, where $\mathbf{x}$ is a vector whose size is shown on the vertical axis.

3.7 Examples of Usage

Table I shows nine examples of different operations in MATLAB and in Parallel Colt. Since Java is a statically typed language, all variable names (along with their types) must be explicitly declared. MATLAB, on the other hand, is a dynamically typed language so there is no need to declare anything. An assignment statement binds a name to an object of any type and later the same name may be assigned to an object of a different type. This feature makes MATLAB expressions generally much more concise than the corresponding expressions in Java. Another essential difference between MATLAB and Parallel Colt arises due to a lack of operator overloading in Java (compare a matrix times vector expressions). Aside from these two differences, the expressions in Table I show that the same level of abstraction is used in MATLAB and Parallel Colt.

Description	MATLAB	Parallel Colt
New 2D dense matrix A	A = zeros(10, 10);	DoubleMatrix2D A = new DenseDoubleMatrix2D(10,10);
Copy of A	B = A;	DoubleMatrix2D B = A.copy();
Transpose of A	B = A';	DoubleMatrix2D B = A.viewDice();
Matrix times vector	B=A*x;	DoubleMatrix2D B = A.zMult(x);
2D FFT of A	B = fft2(A);	DComplexMatrix2D B = A.getFft2();
FFT along columns of A	B = fft(A,2);	DComplexMatrix2D B = A.getFftColumns();
Cosine of A (in-place)	A = cos(A);	A.assign(DoubleFunctions.cos);
Sum all entries of A	s = sum(A(:));	double s = A.zSum();
Location of max of A	[i, j] = ind2sub(size(A), ... find(A == max(A(:))));	double[] max = A.getMaxLocation();

Table I. Comparison of MATLAB and Parallel Colt expressions for a sample set of matrix operations.

4. ITERATIVE IMAGE DEBLURRING

4.1 Basics of Image Deblurring

In applications such as astronomy, medicine, physics and biology, scientists use digital images to record and analyze results from experiments. Environmental effects and imperfections in the imaging system can cause the recorded images to be degraded by blurring and noise. Image restoration (sometimes known as deblurring or deconvolution) is the process of reconstructing or estimating the true image from the degraded one. Image deblurring algorithms can be classified into two types: spectral filtering methods and iterative methods. Algorithms can also be classified into methods that do not require any information about the blur (also called blind deconvolution algorithms) and methods that need that information. In this article we only discuss the latter ones. Information about the blur is usually given in the form of a point spread function (PSF). A PSF is an image that describes the response of an imaging system to a point object. A theoretical PSF can be obtained based on the optical properties of the imaging system. The main advantage of this approach is that the obtained PSF is noise-free. The experimental technique, on the other hand, relies on taking a picture of a point object, for example in astronomy this can be a distant star.

Mathematically image deblurring is the process of computing an approximation of a vector $\mathbf{x}_{\text{true}}$ (which represents the true image scene) from the linear inverse problem

$$\mathbf{b} = \mathbf{A}\mathbf{x}_{\text{true}} + \boldsymbol{\eta}\,. \tag{1}$$

Here, $\mathbf{A}$ is a large, usually ill-conditioned matrix defined by the PSF, $\boldsymbol{\eta}$ is a vector that models additive noise, and $\mathbf{b}$ is a vector representing the recorded image, which is degraded by blurring and noise. We assume that the PSF, and hence $\mathbf{A}$, is known, but the noise is unknown. Because $\mathbf{A}$ is usually severely ill-conditioned, some form of *regularization* needs to be incorporated [Hansen 1997; Vogel 2002]. For example, many regularization methods compute solutions of the form $\mathbf{x}_{\text{reg}} = \mathbf{A}_r^\dagger\mathbf{b}$, where $\mathbf{A}_r^\dagger$ can be thought of as a regularized pseudo-inverse of $\mathbf{A}$. The precise form of $\mathbf{A}_r^\dagger$ depends on many things, including the regularization method, the data $\mathbf{b}$, and the blurring matrix $\mathbf{A}$ [Hansen et al. 2006]. Note that

$$\mathbf{x}_{\text{reg}} = \mathbf{A}_r^\dagger\mathbf{b} = \mathbf{A}_r^\dagger\mathbf{A}\mathbf{x}_{\text{true}} + \mathbf{A}_r^\dagger\boldsymbol{\eta}\,,$$

so such regularization methods attempt to balance the desire to have $\mathbf{A}_r^\dagger\mathbf{A} \approx \mathbf{I}$ while at the same time keeping $\mathbf{A}_r^\dagger\boldsymbol{\eta}$ from becoming too large.

In this section we highlight the advantages and disadvantages of spectral and iterative algorithms to compute regularized approximations of $\mathbf{x}_{\text{true}}$.

Spectral filtering methods exploit structure of the matrix to efficiently compute the singular (or spectral) value decomposition of $\mathbf{A}$, and use this information to construct $\mathbf{A}_r^\dagger$. The spectral filtering algorithms include many well known techniques for image deblurring such as the Wiener filter [Gonzalez and Wintz 1977] and the pseudo inverse filter. But general approaches, such as truncated spectral decompositions and Tikhonov regularization [Hansen et al. 2006] also belong to this group. Whether or not these techniques work well depends on special structure of the PSF, and hence of $\mathbf{A}$. For example, if FFT based methods are used (e.g., Wiener filter [Gonzalez and Wintz 1977]), then there is an implicit assumption that the blur is spatially invariant and that the original image scene is periodic

(the so-called periodic boundary condition). In this case, the matrix $\mathbf{A}$ is circulant, and it is well known that the eigenvalues and eigenvectors of such a matrix can be computed efficiently using FFTs [Hansen et al. 2006].

Other fast transforms, such as the discrete cosine transform (DCT) and the discrete sine transform (DST) can be used for other boundary conditions, but again these approaches only make sense if the blur is spatially invariant. Furthermore, in the case of using fast DCT and DST based methods, the PSF should be symmetric (as in the case of atmospheric turbulence). The advantage of using spectral filtering algorithms is that they can be very efficient, and they are fairly easy to implement. In our recent work [Wendykier and Nagy 2008] we described a multithreaded Java package for spectral deconvolution.

However, there are many limitations of spectral filtering algorithms. First, efficient implementation requires the blur to have a very special structure, and this almost always means spatially invariant. In the case of spatially variant blurs, FFT, DCT and DST based methods do not provide the right basis to use in filtering algorithms. It is possible to generalize the filtering ideas, using the singular value decomposition (SVD), but generally these approaches are very expensive. One exception is if the space variant blur is separable (i.e., the blurring operation can be separated into components involving a single vertical and a single horizontal blur). In this case the matrix $\mathbf{A}$ can be represented as a Kronecker product of two smaller matrices. Another limitation of spectral filtering methods is that it is not possible to include additional constraints, such as nonnegativity, in the reconstruction algorithms. Figure 3 shows a comparison between the spectral and iterative methods - in practice the quality of reconstruction is usually much better when an iterative algorithm is used.

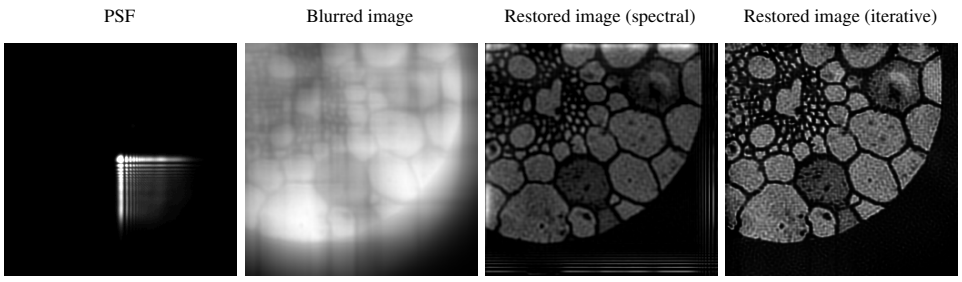

Fig. 3. Spectral vs. iterative image deblurring.

With iterative methods, a sequence of approximations is constructed, where hopefully subsequent approximations provide better reconstructions. Mathematically this is equivalent to solving a particular optimization problem involving $\mathbf{A}$ and $\mathbf{b}$, which could be formulated as something simple like a least squares problem, or something more complicated that incorporates (possibly nonlinear) constraints. As with spectral filtering methods, regularization must be incorporated using, for example, *a priori* constraints, or through appropriate convergence criteria, or even a combination of such techniques.

Well known examples of iterative image reconstruction algorithms include expectation maximization (EM) type approaches (such as the Richardson-Lucy algorithm), conjugate gradient (CG) type methods, and many others. One important advantage of using iterative algorithms is that they can be used on a much wider class of blurring models, including

spatially variant blurs. Although iterative methods are generally more expensive than spectral filtering methods for simple spatially invariant blurs, they are much more efficient for difficult spatially variant blurs. Moreover, it is much easier to incorporate constraints (e.g., nonnegativity) in the algorithms. The main disadvantages of iterative methods are determining how to incorporate regularization (to stabilize the iterative method in the presence of noise), and determining an appropriate stopping iteration. Chung and Nagy in their recent work [Chung et al. 2008] have developed a hybrid conjugate gradient type iterative method that automatically chooses regularization parameters and implements stopping criteria based on the generalized cross validation (GCV) scheme [Hansen et al. 2006].

4.2 RestoreTools

MATLAB's Image Processing Toolbox contains some methods for image restoration, but these have several limitations. For example, they cannot be used with spatially variant blurs. The RestoreTools [Nagy et al. 2004] package contains several additional, modern algorithms which have been studied in the inverse problems and numerical analysis literature. In addition, an object oriented design allows users to easily incorporate our efficient computational kernels in their own algorithms. The package includes 2D and 3D iterative methods for unsymmetric (CGLS [Björck 1996], HyBR [Chung et al. 2008]) and symmetric blurs (MR2 [Hanke 1995]), as well as an algorithm that enforces nonnegativity constraints (MRNSD [Nagy and Strakos 2000]). To accelerate convergence of iterative methods we provide preconditioners (with automatic choice of certain tolerances) based on FFT, DCT and SVD. All the algorithms work for both spatially invariant as well as spatially variant blurs. Moreover, three types of boundary conditions (zero, periodic and reflexive) can be used in a reconstruction. Recently, we have also added support for single precision data and for GPU computations by using AccelerEyes' Jacket toolbox. There are two limitations in the current release: no graphical user interface and no support for color images.

4.3 Iterative Deconvolve 3D

Iterative Deconvolve 3D [Dougherty 2005] is an ImageJ plugin written by Robert Dougherty, OptiNav Inc. It is an implementation of nonnegatively constrained Landweber iteration, where a regularized Wiener filter is used as a preconditioner. Besides the fact that the code is sequential, this plugin has two limitations. First of all, it requires a PSF image to be centered in the field of view. Moreover, it uses a Discrete Hartley Transform (DHT) that works only when the size of the data is a of power-of-two. This means that a blurred image and a PSF need to be padded to the next power-of-two size before processing. When the FFT or DHT are used for image deblurring, padding is always necessary (to avoid wrap effects), but it is always enough to pad each side of a blurred image with an amount that is only half of the size of the PSF image. This property is not implemented in Iterative Deconvolve 3D. Instead, the size of the PSF is disregarded and the blurred image is always padded to the next power-of-two size that is at least 1.5 times larger than the original image. Since usually a blurred image is much larger than the PSF image, this type of padding results not only in very poor performance but it also requires much more memory.

4.4 Parallel Iterative Deconvolution

Parallel Iterative Deconvolution [Wendykier 2008c] is an ImageJ plugin that implements MRNSD, CGLS, HyBR and Landweber algorithms. The first three methods are derived

from RestoreTools and the Landweber algorithm is a parallel version of Iterative Deconvolve 3D with some enhancements. In particular, we have fixed the two aforementioned limitations of the original plugin. Parallel Colt is used as a computational engine for all implemented algorithms. In addition to all the features of RestoreTools (except DCT and SVD preconditioners) the plugin supplies the following options: threshold (the smallest nonnegative pixel value assigned to the restored image), resizing (size of padding: auto, minimal or the next power of two), different output types (same as source, byte, short or float), show iterations, the number of threads and batch processing (can be called from ImageJ macro). All this functionality is enclosed in a clear and intuitive GUI (Fig. 4), where only options that are common for all methods are displayed in the main window (the advanced preferences are hidden under the "Options" button). In a typical usage scenario, there is no need to change the advanced preferences, since the default values are usually optimal. In contrast, Dougherty's Iterative Deconvolve 3D GUI shows all available options in one window which may discourage less experienced users.

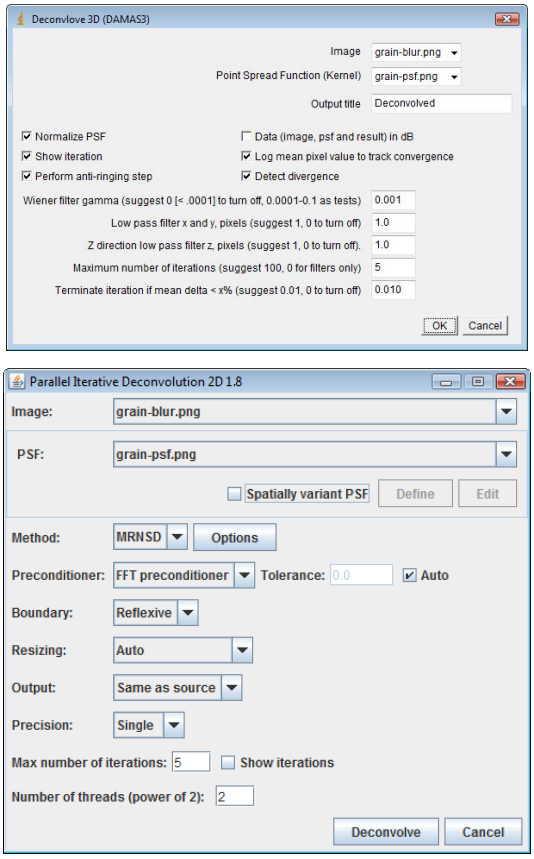

Fig. 4. GUI for Iterative Deconvolve 3D and Parallel Iterative Deconvolution

4.5 Benchmarks

In this subsection we compare the performace of native MATLAB, MATLAB with Jacket, and Java with Parallel Colt. We present benchmark results for FFTs and for iterative image deblurring. As a testbed for our benchmarks we used a machine equipped with two Quad-Core Intel Xeon E5472 processors, 32GB RAM, and NVidia Quadro FX 5600. The system was running Ubuntu Linux 8.10 (64-bit), NVidia CUDA 2.1 Beta, MATLAB R2008b, AccelerEyes Jacket 0.6, Sun Java 1.6.0_10 (64-Bit Server VM) and ImageJ 1.42a. The following Java options were used: `-d64 -server -Xms10g -Xmx10g -XX:+UseParallelGC`.

4.5.1 *FFT.* We have benchmarked single precision FFTs of native MATLAB, MATLAB with Jacket, and Parallel Colt. The timings in Table II and III are an average among 100 calls of each transform. They do not incorporate the "warm up" phase (first two calls require more time) for MATLAB and Parallel Colt. Benchmark results for Jacket include the time required for data transfer to and from the GPU memory. Maximum 8 threads were used in Parallel Colt. It should be noted that the amount of GPU memory is a serious limitation for large-scale problems. On the hardware available, the largest sizes of a matrices that fit into GPU memory were 4096×4096 and $256 \times 256 \times 256$.

Size	MATLAB	MATLAB + Jacket	Parallel Colt
2000×2000	0.16	0.11	0.09
2048×2048	0.45	0.15	0.09
4000×4000	0.75	0.56	0.34
4096×4096	1.99	0.67	0.34
8000×8000	4.22	-	2.47
8192×8192	8.17	-	1.53

Table II. Average execution times (in seconds) for 2D FFT

In the 2D case we observed a speedup of 3 (for 4096×4096) when comparing native MATLAB and MATLAB with Jacket. For 3D transforms the largest speedup was about 3.7 (for $256 \times 256 \times 256$). Nonetheless, Parallel Colt was faster for all tested matrices.

Size	MATLAB	MATLAB + Jacket	Parallel Colt
$100 \times 100 \times 100$	0.04	0.03	0.03
$128 \times 128 \times 128$	0.21	0.06	0.05
$200 \times 200 \times 200$	0.41	0.27	0.22
$256 \times 256 \times 256$	2.00	0.54	0.33
$500 \times 500 \times 500$	8.33	-	4.47
$512 \times 512 \times 512$	16.53	-	2.10

Table III. Average execution times (in seconds) for 3D FFT

4.5.2 *Deconvolution.* The test image for 2D benchmarks[3] is a picture of Ed White performing the first U.S. spacewalk in 1965 [NASA 1965]. The true image is of the size 2048×2048 pixels. The blurred image was generated by reflexive padding of the true data to size 3072×3072, convolving it with Gaussian blur PSF (128×128 pixels, standard deviation = 10), adding 1% white noise and then cropping the resulting image to the size of 2048×2048 pixels. Only preconditioned algorithms have been benchmarked. Table IV presents average execution times (among ten calls) required to perform five iterations in single precision. In the case of Java code, one should observe a significant speedup, especially from 1 to 2 threads. The main reason why the algorithms do not scale linearly when the number of threads is increased is that the size of the image is too small. However, this is the largest image that can be deblurred when using Jacket - larger data caused an error message. In comparison with native MATLAB, Java code is slower only for a single thread. On the other hand, MATLAB code does not scale at all while the number of threads is increased. This shows how poorly MATLAB currently supports multithreaded computations. Once the computations in MATLAB were performed on the GPU, we observed the performace comparable to 8 threads in Java. We have also tested two Java implementations of the Landweber algorithm. Our implementation outperforms Iterative Deconvolve 3D plugin by over 55 times (for 8 threads).

Method	1 thread	2 threads	4 threads	8 threads
CGLS (Java)	29.43	15.99	9.81	7.81
CGLS (MATLAB)	18.52	18.26	18.06	18.16
CGLS (MATLAB+ Jacket)	9.65	-	-	-
MRNSD (Java)	31.30	17.50	10.51	8.48
MRNSD (MATLAB)	19.95	19.68	19.48	19.34
MRNSD (MATLAB+ Jacket)	10.57	-	-	-
HyBR (Java)	33.06	17.93	11.29	9.15
HyBR (MATLAB)	20.64	20.58	19.96	20.13
HyBR (MATLAB+ Jacket)	11.89	-	-	-
Landweber (Java)	9.04	5.07	2.93	2.20
Iterative Deconvolve 3D (Java)	122.87	-	-	-

Table IV. Average execution times (in seconds) for 2D deblurring (image size: 2048 x 2048)

For 3D deblurring we used exactly the same hardware and software. This time the test image is a T1 weighted MRI image of Jeff Orchard's head [Orchard 2007]. The true image is of the size $128 \times 256 \times 256$. The blurred image was generated by zero padding of the true data to size $128 \times 512 \times 512$, convolving it with a Gaussian blur PSF ($16 \times 16 \times 16$ pixels, standard deviation = 1), adding 1% white noise and then cropping the resulting image to the size of $128 \times 256 \times 256$ pixels. In Table V, we have collected all timings. We measured an average execution time required to perform five iterations of preconditioned methods in single precision. Similar to 2D deblurring, Java is slower then

[3]Since the purpose of this section is to compare computational efficiency and not the quality of the image reconstructions, to conserve space we do not include any pictures. Interested readers can find the test and reconstructed images discussed in this section at [Wendykier 2008c].

MATLAB only for a single thread and we observed no advantage of using multiple threads in MATLAB. The performace of GPU computations is comparable to the best results in Java. The Landweber algorithm in Parallel Iterative Deconvolution is over 44 times faster than Iterative Deconvolve 3D.

Method	1 thread	2 threads	4 threads	8 threads
CGLS (Java)	91.12	46.59	28.98	21.96
CGLS (MATLAB)	53.63	53.68	53.13	53.40
CGLS (MATLAB+ Jacket)	21.79	-	-	-
MRNSD (Java)	97.05	49.78	30.43	23.15
MRNSD (MATLAB)	56.77	57.78	56.20	56.51
MRNSD (MATLAB+ Jacket)	23.85	-	-	-
HyBR (Java)	98.98	53.10	33.84	25.83
HyBR (MATLAB)	58.41	58.66	57.35	57.06
HyBR (MATLAB+ Jacket)	26.03	-	-	-
Landweber (Java)	25.88	13.64	8.13	5.66
Iterative Deconvolve 3D	250.05	-	-	-

Table V. Average execution times (in seconds) for 3D deblurring

5. SUMMARY

We have demonstrated the advantage of exploiting available hardware on modern, and affordable, computer architectures in scientific computing with multithreaded programming in Java and with general purpose computation on GPUs in MATLAB. A significant contribution of our work is Parallel Colt, a multithreaded Java library for image processing. We have also described an ImageJ plugin for iterative image deblurring as a typical application of this library. In addition, by using AccelerEyes' Jacket toolbox to access the computer's GPU, we were able to obtain significant speed up of MATLAB computations. Thus, we are able to provide Java and MATLAB software to solve a class of problems that are ubiquitous in real image processing applications, and which can effectively make use of multi-core CPUs and powerful GPUs available on affordable desktop and laptop computers.

Our results show that Java can be a competitive language for certain scientific computing applications. In addition, we observe that MATLAB currently does not effectively take advantage of multi-core architectures (most notably for computing FFTs), though we hope this will change in future releases. Finally, we note that although we obtained significant speedup when using MATLAB with Jacket, we were limited in the size of images that could be processed, and there are accuracy concerns associated with the FFTs used by CUDA (and hence Jacket).

REFERENCES

ACCELERYES. 2008. Jacket. http://accelereyes.com/.

AHMED, N., NATARAJAN, T., AND RAO, K. R. 1974. Discrete cosine transfom. *Transactions on Computers C-23,* 1, 90–93.

AMERDO, B., BODNARTCHOUK, V., CAROMEL, D., DELBÉ, C., HUET, F., AND TABOADA, G. L. 2008. Current State of Java for HPC. Tech. Rep. inria-00312039, INRIA.

ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, L. S., DEMMEL, J., DONGARRA, J. J., CROZ, J. D., HAMMARLING, S., GREENBAUM, A., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

BAOSHE ZHANG. 2005. Java FFTPack Project. http://jfftpack.sourceforge.net/.

BARRETT, R., BERRY, M., CHAN, T. F., DEMMEL, J., DONATO, J., DONGARRA, J., EIJKHOUT, V., POZO, R., ROMINE, C., AND DER VORST, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

BJÖRCK, Å. 1996. *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, Penn.

BJØRN-OVE HEIMSUND. 2007. Matrix Toolkits for Java. http://ressim.berlios.de/.

BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. 2002. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software 28,* 2 (June), 135–151.

BLUESTEIN, L. I. 1968. A linear filtering approach to the computation of the discrete fourier transform. *Northeast Electronics Research and Engineering Meeting Record* 10, 218–219.

BUTTARI, A., LANGOU, J., KURZAK, J., AND DONGARRA, J. 2007. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. Tech. rep., Innovative Computing Laboratory.

BYOUS, J. 2003. Java Technology: The Early Years. http://java.sun.com/features/1998/05/birthday.html.

CHUNG, J., NAGY, J. G., AND P'LEARY, D. 2008. A Weighted GCV Method for Lanczos Hybrid Regularization. *Elec. Trans. Numer. Anal. 28*, 149–167.

COOLEY, J. W. AND TUKEY, J. W. 1965. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation 19,* 90, 297–301.

DOOLIN, D. M., DONGARRA, J., AND SEYMOUR, K. 1999. JLAPACK - compiling LAPACK Fortran to Java. *Sci. Program. 7,* 2, 111–138.

DOUGHERTY, R. 2005. Extensions of DAMAS and Benefits and Limitations of Deconvolution in Beamforming. In *11th AIAA/CEAS Aeroacoustics Conference*.

DRA SYSTEMS. 2000. OR-Objects. http://opsresearch.com/OR-Objects/.

FRIGO, M. AND JOHNSON, S. G. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE 93,* 2, 216–231.

GONZALEZ, R. C. AND WINTZ, P. 1977. *Digital Image Processing*. Addison-Wesley Pub. Co, Chapter 5.

HANKE, M. 1995. *Conjugate gradient type methods for ill-posed problems*. Pitman Research Notes in Mathematics, Longman Scientific & Technical, Harlow, Essex.

HANSEN, P. C. 1997. *Rank-deficient and discrete ill-posed problems*. SIAM, Philadelphia, PA.

HANSEN, P. C., NAGY, J. G., AND O'LEARY, D. P. 2006. *Deblurring Images Matrices, Spectra and Fitering*. SIAM.

HARTLEY, R. 1942. A more symmetrical Fourier analysis applied to transmission problems. In *Proceedings of IRE*.

HICKLIN, J., MOLER, C., WEBB, P., BOISVERT, R. F., MILLER, B., POZO, R., AND REMINGTON, K. 2005. JAMA : A Java Matrix Package. http://math.nist.gov/javanumerics/jama/.

HOSCHEK, W. 2000. Versatile and Efficient Dense and Sparse Multi-Dimensional Arrays.

HOSCHEK, W. 2004. Colt Project. http://dsd.lbl.gov/%7Ehoschek/colt/index.html.

JAVAGL. 2008a. JCublas. http://javagl.de/jcuda/jcublas/JCublas.html.

JAVAGL. 2008b. JCufft. http://javagl.de/jcuda/jcufft/JCufft.html.

JEAN-MARIE DAUTELLE. 2007. JScience. http://jscience.org/.

NAGY, J. G., PALMER, K., AND PERRONE, L. 2004. Iterative methods for image deblurring: A matlab object-oriented approach. *Numerical Algorithms 36,* 1 (May), 73–93.

NAGY, J. G. AND STRAKOS, Z. 2000. Enforcing Nonnegativity in Image Reconstruction Algorithms. In *Mathematical Modeling, Estimation and Imaging*, D. W. et. al., Ed. Vol. 4121. 182–190.

NASA. 1965. Great Images in NASA. Ed White performs first U.S. spacewalk. http://grin.hq.nasa.gov/ABSTRACTS/GPN-2006-000025.html.

NVIDIA CORPORATION. 2008. CUDA Zone. http://www.nvidia.com/object/cuda_home.html.

OOURA, T. 2006. General Purpose FFT (Fast Fourier/Cosine/Sine Transform) Package. http://www.kurims.kyoto-u.ac.jp/%7Eooura/fft.html.

ORCHARD, J. 2007. His Brain. http://www.cs.uwaterloo.ca/%7Ejorchard/mri/.

P. N. SWARZTRAUBER. 2004. FFTPACK5. http://www.cisl.ucar.edu/css/software/fftpack5/.

RASBAND, W. S. 2008. ImageJ, U. S. National Institutes of Health, Bethesda, Maryland, USA. http://rsb.info.nih.gov/ij/.

SCHATZMAN, J. C. 1996. Accuracy of the discrete fourier transform and the fast Fourier transform. *SIAM Journal on Scientific Computing 17,* 5, 1150–1166.

SUN MICROSYSTEMS. 2004. New Features and Enhancements J2SE 5.0. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html.

VISUAL NUMERICS. 2008. JMSL Numerical Library for Java. http://www.vni.com/products/imsl/jmsl/jmsl.php.

VOGEL, C. R. 2002. *Computational Methods for Inverse Problems*. SIAM, Philadelphia, PA.

W. KAHAN AND J. D. DARCY. 1998. How Java's Floating-Point Hurts Everyone Everywhere. http://www.cs.berkeley.edu/%7Ewkahan/JAVAhurt.pdf.

WENDYKIER, P. 2008a. JTransforms Project. http://piotr.wendykier.googlepages.com/jtransforms.

WENDYKIER, P. 2008b. Parallel Colt Project. http://piotr.wendykier.googlepages.com/parallelcolt.

WENDYKIER, P. 2008c. Parallel Itarative Deconvolution. http://piotr.wendykier.googlepages.com/iterativedeconvolution.

WENDYKIER, P. AND NAGY, J. G. 2008. Large-scale image deblurring in java. In *Computational Science - ICCS 2008, 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part I*. 721–730.

WHALEY, R. C. AND DONGARRA, J. J. 1998. Automatically Tuned Linear Algebra Software (ATLAS). In *Proceedings of Supercomputing 1998*.

YIP, P. AND RAO, K. R. 1980. A Fast Computational Algorithm for the Discrete Sine Transform. *IEEE Trans. Commun. 28,* 2.