Technical Report

TR-2008-002

Large-Scale Image Deblurring in Java

by

Piotr Wendykier, James Nagy

Mathematics and Computer Science

EMORY UNIVERSITY

Large-Scale Image Deblurring in Java

Piotr Wendykier and James G. Nagy*

Dept. of Math and Computer Science, Emory University, Atlanta GA, USA,
piotr.wendykier@emory.edu, nagy@mathcs.emory.edu

Abstract. This paper describes *Parallel Spectral Deconvolution* (PSD) Java software for image deblurring. A key component of the software, *JTransforms*, is the first, open source, multithreaded FFT library written in pure Java. Benchmarks show that JTransforms is competitive with current C implementations, including the well-known FFTW package. Image deblurring examples, including performance comparisons with existing software, are also given.

1 Motivation

Instruments that record images are integral to advancing discoveries in science and medicine – from astronomical investigations, to diagnosing illness, to studying bacterial and viral diseases [1,2,3]. Computational science has an important role in improving image quality through the development of post-processing image reconstruction and enhancement algorithms and software. Probably the most commonly used post-processing technique is image deblurring, or deconvolution [4]. Mathematically this 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 that models the blurring operation, $\boldsymbol{\eta}$ is a vector that models additive noise, and $\mathbf{b}$ is a vector representing the recorded image, which is degraded by blurring and noise.

Generally, it is assumed that the blurring matrix $\mathbf{A}$ is known (at least implicitly), but the noise is unknown. Because $\mathbf{A}$ is usually severely ill-conditioned, some form of *regularization* needs to be incorporated [5,6]. Many regularization methods, including Tikhonov, truncated singular (or spectral) value decomposition (TSVD), and Wiener filter, 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 $\mathbf{A}$ [4]. The actual implementation of computing $\mathbf{x}_{\text{reg}}$ can often be done very efficiently using fast Fourier transforms (FFT) and fast discrete cosine transforms (DCT).

This paper describes our development of *Parallel Spectral Deconvolution* (PSD) [7] Java software for image deblurring, including a plugin for the open source image processing system, ImageJ [8]. A key component of our software is the first, open source, multithreaded FFT library written in pure Java, which we call *JTransforms* [7].

* Research supported by the NSF under grant DMS-05-11454.

This paper is organized as follows. In Section 2 we describe some basic image deblurring algorithms, and how fast transforms, such as FFTs and DCTs, can be used for efficient implementations. Section 3 describes the performance of our Java implementations, with a particular focus on JTransforms. Benchmarks show that our multithreaded Java approach is competitive with current C implementations, including the well-known FFTW package [9]. Image deblurring examples, including performance comparisons with existing software, are also given.

2 Deblurring Techniques

The deblurring techniques considered in this paper are based on filtering out certain spectral coefficients of the computed solution.

2.1 Regularization by Filtering

We begin by showing why regularization is needed, and how it can be done through spectral filtering. To simplify the discussion, we assume $\mathbf{A}$ is an $n \times n$ *normal* matrix [10], meaning that it has a spectral value decomposition (SVD)[1]

$$\mathbf{A} = \mathbf{Q}^* \boldsymbol{\Lambda} \mathbf{Q}$$

where $\boldsymbol{\Lambda}$ is a diagonal matrix containing the eigenvalues of $\mathbf{A}$, $\mathbf{Q}$ is a matrix whose columns, $\mathbf{q}_i$, are the corresponding eigenvectors, $\mathbf{Q}^*$ is the complex conjugate transpose of $\mathbf{Q}$, and $\mathbf{Q}^*\mathbf{Q} = \mathbf{I}$. We assume further that the eigenvalues are ordered so that $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n| \geq 0$. Using the spectral decomposition, the inverse solution of equation (1) can be written as

$$\mathbf{x}_{\text{inv}} = \mathbf{A}^{-1}\mathbf{b} = \mathbf{A}^{-1}(\mathbf{A}\mathbf{x}_{\text{true}} + \boldsymbol{\eta}) = \mathbf{x}_{\text{true}} + \mathbf{A}^{-1}\boldsymbol{\eta} = \mathbf{x}_{\text{true}} + \sum_{i=1}^{n} \frac{\widehat{\eta}_i}{\lambda_i}\, \mathbf{q}_i \,,$$

where $\widehat{\boldsymbol{\eta}} = \mathbf{Q}^*\boldsymbol{\eta}$. That is, the inverse solution is comprised of two terms: the desired true solution and an error term caused by noise in the data. To understand why the error term usually dominates the inverse solution, it is necessary to know the following properties of image deblurring [4,5]:

- Assuming the problem is scaled so that $|\lambda_1| = 1$, the eigenvalues, $|\lambda_i|$, decay to, and cluster at 0, without a significant gap to indicate numerical rank.
- The eigenvectors $\mathbf{q}_i$ corresponding to small $|\lambda_i|$ tend to have more oscillations than the eigenvectors corresponding to large $|\lambda_i|$.

These properties imply that the high frequency components in the error are highly magnified by division of small eigenvalues. The computed inverse solution is dominated by

[1] We realize that "SVD" usually refers to "singular value decomposition". We do not think there should be any confusion because our discussion of filtering can be done using the singular value decomposition in place of the spectral value decomposition.

these high frequency components, and is in general a very poor approximation of the true solution, $\mathbf{x}_{\text{true}}$.

In order to compute an accurate approximation of $\mathbf{x}_{\text{true}}$, or at least one that is not horribly corrupted by noise, the solution process must be modified. This process is usually referred to as *regularization* [5,6]. One class of regularization methods, called *filtering*, can be formulated as a modification of the inverse solution [5]. Specifically, a filtered solution is defined as

$$\mathbf{x}_{\text{reg}} = \mathbf{A}_r^{\dagger}\mathbf{b} \tag{2}$$

where $\mathbf{A}_r^{\dagger} = \mathbf{Q}^*\text{diag}\left(\frac{\phi_1}{\lambda_1}, \frac{\phi_2}{\lambda_2}, \ldots, \frac{\phi_n}{\lambda_n}\right)\mathbf{Q}$. The *filter factors*, ϕ_i, satisfy $\phi_i \approx 1$ for large $|\lambda_i|$, and $\phi_i \approx 0$ for small $|\lambda_i|$. That is, the large eigenvalue (low frequency) components of the solution are reconstructed, while the components corresponding to the small eigenvalues (high frequencies) are filtered out. Different choices of filter factors lead to different methods; popular choices are the truncated SVD (or pseudo-inverse), Tikhonov, and Wiener filters [5,6,11].

2.2 Tikhonov Filtering

To illustrate spectral filtering, consider the Tikhonov regularization filter factors

$$\phi_i = \frac{|\lambda_i|^2}{|\lambda_i|^2 + \alpha^2}\,, \tag{3}$$

where the scalar α is called a regularization parameter, and usually satisfies $|\lambda_n| \leq \alpha \leq |\lambda_1|$. Note that smaller α lead to more ϕ_i approximating 1.

The regularization parameter is problem dependent, and in general it is nontrivial to choose an appropriate value. Various techniques can be used, such as the discrepancy principle, the L-curve, and generalized cross validation (GCV) [5,6]. There are advantages and disadvantages to each of these approaches [12], especially for large-scale problems. In this work we use GCV, which, using the SVD of $\mathbf{A}$, requires finding α to minimize the function

$$G(\alpha) = n\sum_{i=1}^{n}\left(\frac{\alpha^2\,|\widehat{b}_i|}{|\lambda_i|^2 + \alpha^2}\right)^2 \Bigg/ \left(\sum_{i=1}^{n}\frac{\alpha^2}{|\lambda_i|^2 + \alpha^2}\right)^2\,, \tag{4}$$

where $\widehat{\mathbf{b}} = \mathbf{Q}^*\mathbf{b}$. Standard optimization routines can be used to minimize $G(\alpha)$.

Tikhonov filtering, and using GCV to choose regularization parameters, has proven to be effective for a wide class of inverse problems. Unfortunately for large scale problems such as image deblurring, it may not be computationally feasible to compute the SVD of $\mathbf{A}$. One way to overcome this difficulty is to exploit structure in the problem.

2.3 Fast Transform Filters

In image deblurring, $\mathbf{A}$ is a structured matrix that describes the blurring operation, and is given implicitly in terms of a point spread function (PSF). A PSF is an image of a

point source object, and provides the essential information to construct $\mathbf{A}$. The structure of $\mathbf{A}$ depends on the PSF and on the imposed boundary condition [4]. In this subsection we describe two structures that arise in many image deblurring problems. However, due to space limitations, we cannot provide complete details; the interested reader should see [4] for more information.

If the blur is assumed to be *spatially invariant* then the PSF is the same regardless of the position of the point source in the image field of view. In this case, if we also enforce periodic boundary conditions, then $\mathbf{A}$ has a circulant matrix structure, and the spectral factorization

$$\mathbf{A} = \mathbf{F}^* \boldsymbol{\Lambda} \mathbf{F}\,, \tag{5}$$

where $\mathbf{F}$ is a discrete Fourier transform (DFT); a d-dimensional image implies $\mathbf{F}$ is a d-dimensional DFT matrix. In this case, the matrix $\mathbf{F}$ does not need to be constructed explicitly; a matrix vector multiplication $\mathbf{Fb}$ is equivalent to computing a DFT of $\mathbf{b}$, and similarly $\mathbf{F}^*\mathbf{b}$ is equivalent to computing an inverse DFT. Efficient implementations of DFTs are usually referred to as fast Fourier transforms (FFT). The eigenvalues of $\mathbf{A}$ can be obtained by computing an FFT of the first column of $\mathbf{A}$, and the first column of $\mathbf{A}$ can be obtained directly from the PSF. Thus, the computational efficiency of spectral filtering methods for image deblurring with a spatially invariant PSF and periodic boundary conditions requires efficient FFT routines.

If the image has significant features near the boundary of the field of view, then periodic boundary conditions can cause ringing artifacts in the reconstructed image. In this case it may be better to use reflexive boundary conditions. But changing the boundary conditions changes the structure of $\mathbf{A}$, and it no longer has the Fourier spectral decomposition given in equation (5). However, if the PSF is also symmetric about its center, then $\mathbf{A}$ is a mix of Toeplitz and Hankel structures [4], and has the spectral value decomposition

$$\mathbf{A} = \mathbf{C}^T \boldsymbol{\Lambda} \mathbf{C}\,, \tag{6}$$

where $\mathbf{C}$ is the discrete cosine transform (DCT) matrix; a d-dimensional image implies $\mathbf{C}$ is a d-dimensional DCT matrix. As with FFTs, there are very efficient algorithms for evaluating DCTs. Furthermore, computations such as the matrix vector multiplication $\mathbf{Cb}$ and $\mathbf{C}^T\mathbf{b}$ are done by calling DCT and inverse DCT functions. The eigenvalues of $\mathbf{A}$ can be obtained by computing a DCT of the first column of $\mathbf{A}$, and the first column of $\mathbf{A}$ can be obtained directly from the PSF. Note that in the case of the FFT, $\mathbf{F}$ has complex entries and thus computations necessarily require complex arithmetic. However, in the case of the DCT, $\mathbf{C}$ has real entries, and all computations can be done in real arithmetic.

Efficient FFT and DCT routines are essential for spectral deblurring algorithms. The next section describes our contribution to the development of efficient parallel Java codes for these important problems.

3 Using Java for Image Deblurring

Java is ideally suited to provide efficient, open source image deblurring software that can be used in inexpensive imaging devices for point of care medical applications. Java

implementations are available for virtually all computing platforms, and since May 2007 the source code of Java is distributed under the terms of the GNU General Public License. Moreover, Java has native support for multithreaded programming, which has become a mandatory paradigm in the era of multicore CPUs. Finally, sophisticated imaging functionality is built into Java, allowing for efficient visualization and animation of computational results.

Significant improvements have been made to Java since the 1996 release of JDK 1.0, including Just-In Time compilation, memory allocation enhancements, and utilization of performance features in modern x86 and x64 CPUs [13]. It is no longer the case that Java is too slow for high-performance scientific computing applications; this point is illustrated below for spectral image deblurring.

There are disadvantages to using Java in scientific computing, including 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. Fortunately open source numerical libraries, such as Colt [14], have been developed to overcome these disadvantages. For our work, we are implementing a fully multithreaded version of Colt, which we call Parallel Colt [7].

In the rest of this section we describe Java implementations of JTransforms, ImageJ and associated plugins for image deblurring.

3.1 JTransforms

Fast Fourier Transform. An FFT algorithm is the most efficient method to compute a DFT, with a complexity of $\Theta(N \log(N))$ to compute a DFT of a d-dimensional array containing N components. An FFT algorithm was first proposed by Gauss in 1805 [15], but it was the 1965 work by Cooley and Tukey [16] that is generally credited for popularizing its use. The most common variant of the algorithm, called radix-2, uses a divide-and-conquer approach to recursively split the DFT of size N into two parts of size N/2. Other splittings can be used as well, including mixed-radix and split-radix algorithms [17].

Th split-radix algorithm has the lowest arithmetic operation count to compute a DFT when N is a power of 2 [18]. The algorithm was first described in 1968 by Yavne [19] and then reinvented in 1984 by Duhamel and Hollmann [20]. The idea here is to recursively divide a DFT of size N into one DFT of size N/2 and two DFTs of size N/4. Further details about split-radix algorithm can be found in [17].

Parallel Implementation in Java. JTransforms is the first, open source, multithreaded FFT library written in pure Java. The code was derived from the General Purpose FFT Package (OouraFFT) written by Ooura [21]. OouraFFT is a multithreaded implementation of the split-radix 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. Even so, the package has

several limitations arising from the split-radix algorithm. First of all, the length of the input data has to be a power of two. Second, the number of computational threads must also be a power of 2. Finally, one-dimensional transforms can only use two or four threads.

JTransforms, with few exceptions, share all the features and limitations of Oura's C implementation. However, there are some important distinctions. First, JTransforms uses thread pools, while OouraFFT 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 JTransforms and the OouraFFT is the use of "automatic" multithreading. In JTransforms, threads are used automatically when computations are done on a machine with multiple CPUs. Conversely, both OouraFFT and FFTW require manually setting up the maximum number of computational threads. Lastly, JTransform's API is much simpler than OouraFFT, 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.

The release of Java 5 in 2004 came with a number of significant new language features [22]. One feature that we have found to be very useful is the *cached thread pool*, which creates new threads as needed, and reuses previously constructed threads when they become available. This feature allows to improve the performance of programs that execute many short-lived asynchronous tasks.

Benchmark. To show the performance of JTransforms we have benchmarked the code against the original OouraFFT and also against FFTW 3.1.2. The benchmark was run on the Sun Microsystems SunFire V40z server, with 4 Dual Core AMD Opteron Processors 875 (2.2GHz) and 32 GB of RAM memory. The machine had installed Red Hat Enterprise Linux version 5 (kernel 2.6.18-8.1.14.el5), gcc version 3.4.6 and Java version 1.6.0_03 (64-bit server VM). The following Java options were used: -d64 -server -Xms15g -Xmx15g. For the OouraFFT, we used -O2 flag for the C compiler (one can get slightly better timings with unsafe flags: -O6 - -fast-math). All libraries were set to use a maximum of eight threads and DFTs were computed in-place. The timings in Tables 1 and 2 are an average among 100 calls of each transform. This average execution time does not incorporate the "warm up" phase (the first two calls require more time) for JTransforms and OouraFFT. Similarly, for FFTW, the times do not incorporate the planning phase. Table 1 presents the benchmark results for computing two-dimensional complex forward DFTs. For $2^9 \times 2^9$, $2^{10} \times 2^{10}$ and $2^{12} \times 2^{12}$ sizes, JTransforms outperforms all other tested libraries.

Table 1. Average execution time (milliseconds) for 2-D, complex forward DFT

Library \ Size	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
JTransforms	2.43	3.76	6.21	32.84	198.31	529.81	4028.17	15682.78
OouraFFT	0.74	3.15	12.60	33.66	202.78	789.25	4165.33	16738.65
FFTW_ESTIMATE	1.15	4.84	31.75	131.80	1149.87	2715.39	26889.97	49670.29
FFTW_MEASURE	0.83	2.91	10.73	37.65	182.77	840.09	6665.73	14735.13
FFTW_PATIENT	0.67	2.81	11.73	36.84	179.55	884.39	3761.50	56522.40

Table 2 shows benchmark results for three-dimensional, complex forward DFTs. Once again, our Java implementation is faster than OouraFFT for almost all sizes of input data. Moreover, starting from $2^6 \times 2^6 \times 2^6$, JTransforms is faster than FFTW. More benchmark results including discrete cosine and sine transforms, can be found at the JTransforms website [7].

Table 2. Average execution time (milliseconds) for 3-D, complex forward DFT

Library \ Size	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9
JTransforms	0.12	1.09	2.35	5.02	6.43	46.85	553.21	7115.84
OouraFFT	0.001	0.02	0.15	1.67	11.38	58.63	847.13	12448.24
FFTW_ESTIMATE	0.48	0.39	0.44	1.59	11.18	110.14	1471.14	34326.50
FFTW_MEASURE	0.48	0.37	0.44	1.23	8.28	48.69	601.88	7432.08
FFTW_PATIENT	0.001	0.01	0.10	1.48	8.36	47.27	573.77	8936.34

3.2 Deconvolution Plugins for ImageJ

ImageJ [8] is an open source image processing program written in Java by Wayne Rasband, a researcher working 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.

DeconvolutionJ [23] is an ImageJ plugin written by Nick Linnenbrügger that implements spectral deconvolution based on the Regularized Wiener Filter [11]. The plugin has a number of limitations. It can handle arbitrary-sized two- and three-dimensional images, although it requires the PSF image to be the same size as the blurred image, and it must be centered in the field of view. In addition, the regularization parameter of the Wiener filter must be specified manually and there is no update option to efficiently deblur the same image with different values of the regularization parameter. Last, but not least, DeconvolutionJ is a serial implementation, and therefore cannot take advantage of modern multicore processors.

Our implementation of spectral deconvolution plugin, Parallel Spectral Deconvolution (PSD), does not suffer from any of these limitations. The current version implements Tikhonov- and TSVD-based image deblurring [4]. Our multithreaded approach uses both JTransforms and Parallel Colt, so we were able to achieve a superior performance compared to DeconvolutionJ. PSD's features include two choices of boundary conditions (reflexive and periodic), automatic choice of regularization parameter using GCV, a very fast parameter update option, and the possibility of defining the number of computational threads. By default, the plugin recognizes the number of available CPUs and uses that many threads. Nevertheless, current implementation of PSD has a couple of limitations. First, color images are not supported (DeconvolutionJ is also limited to grayscale images). The second limitation arises due to JTransforms, where the size of input data and the number of threads must be power of two numbers. In order to support images of arbitrary size, PSD uses padding. The number of threads, however, must be a power of two number.

In order to test the performance of PSD, we also used the SunFire V40z with ImageJ version 1.39k. The following Java options were used: -d64 -server -Xms15g -Xmx15g -XX:+UseParallelGC. The test image (see Figure 1) is a picture of Ed White performing first U.S. spacewalk in 1965 [24]. The true image is of the size 4096×4096 pixels. The blurred image was generated by reflexive padding of the true data to size 6144×6144, convolving it with Gaussian blur PSF (standard deviation = 20), adding 1% white noise and then cropping the resulting image to the size of 4096×4096 pixels.

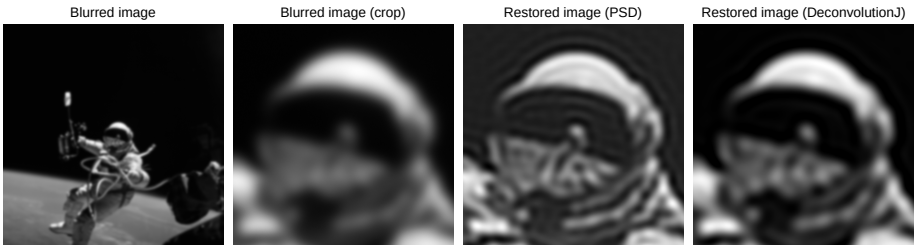

Fig. 1. Astronaut image: blurred and restored data

Figure 1 shows the blurred data as well as the deblurred astronaut images using DeconvolutionJ and PSD. To better illustrate the quality of deblurring, we display a small region of the blurred and reconstructed images. In PSD, we used the Tikhonov method with reflexive boundary conditions and regularization parameter equal 0.01. Similarly, in DeconvolutionJ, we used no resizing (the image size was already a power of two), double precision for complex numbers (PSD uses only double precision) and the same value for the regularization parameter.

Table 3 presents average execution times among 10 calls of each method. All timings are given in seconds and the numbers in brackets include the computation of the regularization parameter. One should notice a significant speedup, especially from 1 to 2 threads. The last row in Table 3 shows the execution time for DeconvolutionJ, which is almost 10 times greater than the worst case of PSD (Tikhonov, FFT, 1 thread) and almost 30 times greater then the best case of PSD (Tikhonov, DCT, 8 threads).

Table 3. Average execution times (in seconds) for 2D deblurring (numbers in brackets include the computation of the regularization parameter)

Method	1 thread	2 threads	4 threads	8 threads
Tikhonov, FFT	18.2 (77.6)	13.2 (51.2)	11.0 (39.3)	10.6 (36.7)
Tikhonov, DCT	15.1 (62.9)	9.9 (41.0)	7.3 (29.5)	6.1 (27.4)
DeconvolutionJ	181.7	-	-	-

For 3D deblurring we used exactly the same hardware and software. This time the test image (see Figure 3), is a T1 weighted MRI image of Jeff Orchard's head [25]. The true image is of the size $128 \times 256 \times 256$ pixels. 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 (standard deviation = 1), adding 1% white noise and then cropping the resulting image to the size of $128 \times 256 \times 256$ pixels.

Table 4. Average execution times (in seconds) for 3D deblurring (numbers in brackets include the computation of the regularization parameter)

Method	1 thread	2 threads	4 threads	8 threads
Tikhonov, FFT	9.5 (31.8)	7.5 (21.4)	7.4 (17.3)	6.7 (15.4)
Tikhonov, DCT	6.4 (29.7)	3.8 (17.4)	2.5 (11.6)	2.0 (10.5)
DeconvolutionJ	31.6	-	-	-

Figure 4 shows the 63^{rd} slice of the deblurred head images. In PSD, we used the Tikhonov method with periodic boundary conditions and regularization parameter equal 0.204301. In DeconvolutionJ, we used exactly the same parameters as for the 2D astronaut image. In Table 4, we have collected all timings. Once again, the execution time for DeconvolutionJ is over 3 times greater than the worst case of PSD (Tikhonov, FFT, 1 thread) and almost 16 times greater that the best case of PSD (Tikhonov, DCT, 8 threads).

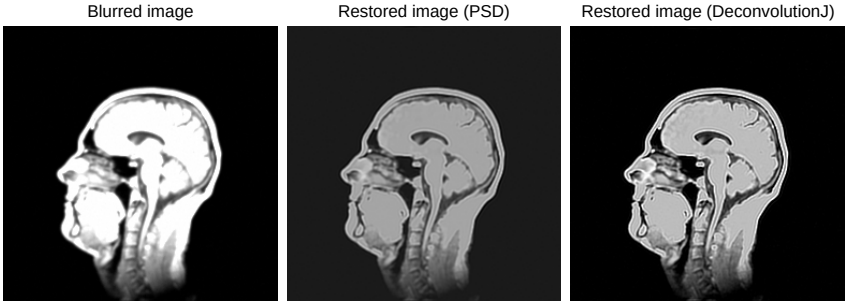

Fig. 2. Head image (63^{rd} slice): blurred and restored data

4 Conclusion

In this paper we have described our research efforts to develop computationally efficient Java software for image deblurring. A key component of this software, JTransforms, is the first, open source, multithreaded FFT library written in pure Java. Due to usage of the cache thread pool we are able to achieve superior performance and speedup on symmetric multiprocessing machines. Numerical results illustrate that our Parallel Spectral Deconvolution package outperforms the ImageJ plugin, DeconvolutionJ, and that our Java FFT implementation, JTransforms, is highly competitive with optimized C implementations, such as FFTW.

References

1. Sarder, P., Nehorai, A.: Deconvolution methods for 3D fluorescence microscopy images. IEEE Signal Proc. Mag. (May 2006) 32–45
2. Roggemann, M.C., Welsh, B.: Imaging Through Turbulence. CRC Press, Boca Raton, FL (1996)
3. Sechopoulos, I., Suryanarayanan, S., Vedantham, S., D'Orsi, C.J., Karellas, A.: Scatter radiation in digital tomosynthesis of the breast. Med. Phys **34** (2007) 564–576
4. Hansen, P.C., Nagy, J.G., O'Leary, D.P.: Deblurring Images: Matrices, Spectra and Filtering. SIAM (2006)
5. Hansen, P.C.: Rank-deficient and discrete ill-posed problems. SIAM, (1997)
6. Vogel, C.R.: Computational Methods for Inverse Problems. SIAM, (2002)
7. Wendykier, P.: JTransforms, Parallel Colt, Parallel Spectral Deconvolution (2007) http://piotr.wendykier.googlepages.com/.
8. Rasband, W.S.: ImageJ, U. S. National Institutes of Health, Bethesda, Maryland, USA (2007). http://rsb.info.nih.gov/ij/.
9. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE **93**(2) (2005) 216–231
10. Stewart, G.W.: Matrix Algorithms, Volume 1: Basic Decompositions. SIAM, (1998)
11. Gonzalez, R.C., Wintz, P.: 5. In: Digital Image Processing. Addison-Wesley (1977)
12. Kilmer, M.E., O'Leary, D.P.: Choosing regularization parameters in iterative methods for ill-posed problems. SIAM J. Matrix Anal. Appl. **22** (2001) 1204–1221
13. Doederlein, O.: Mustang's HotSpot Client gets 58% faster! (2005) http://weblogs.java.net/blog/opinali/archive/2005/11/mustangs_hotspo_1.html.
14. Hoschek, W.: Colt Project (2004) http://dsd.lbl.gov/%7Ehoschek/colt/index.html.
15. Heideman, M.T., Johnson, D.H., Burrus, C.S.: Gauss and the history of the fast Fourier transform. Archive for History of Exact Sciences **34** (1985) 265–277
16. Cooley, J.W., Tukey, J.W.: An Algorithm for the Machine Calculation of Complex Fourier Series. Mathematics of Computation **19**(90) (1965) 297–301
17. Van Loan, C.: Computational Frameworks for the Fast Fourier Transform. SIAM (1992)
18. Johnson, S.G., Frigo, M.: A modified split-radix FFT with fewer arithmetic operations. IEEE Trans. Signal Processing **55**(1) (2007) 111–119
19. Yavne, R.: An economical method for calculating the discrete Fourier transform. In: AFIPS Fall Joint Computer Conference. (1968) 115–125
20. Duhamel, P., Hollmann, H.: Split Radix FFT Algorithms. Electronic Letters **20** (1984) 14–16
21. Ooura, T.: General Purpose FFT (Fast Fourier/Cosine/Sine Transform) Package (2006) http://www.kurims.kyoto-u.ac.jp/%7Eooura/fft.html.
22. Sun Microsystems: New Features and Enhancements J2SE 5.0 (2004) http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html.
23. Linnenbrügger, N.: FFTJ and DeconvolutionJ (2002) http://rsb.info.nih.gov/ij/plugins/fftj.html.
24. NASA: Great Images in NASA. Ed White performs first U.S. spacewalk. (1965) http://grin.hq.nasa.gov/ABSTRACTS/GPN-2006-000025.html.
25. Orchard, J.: His Brain (2007) http://www.cs.uwaterloo.ca/%7Ejorchard/mri/.