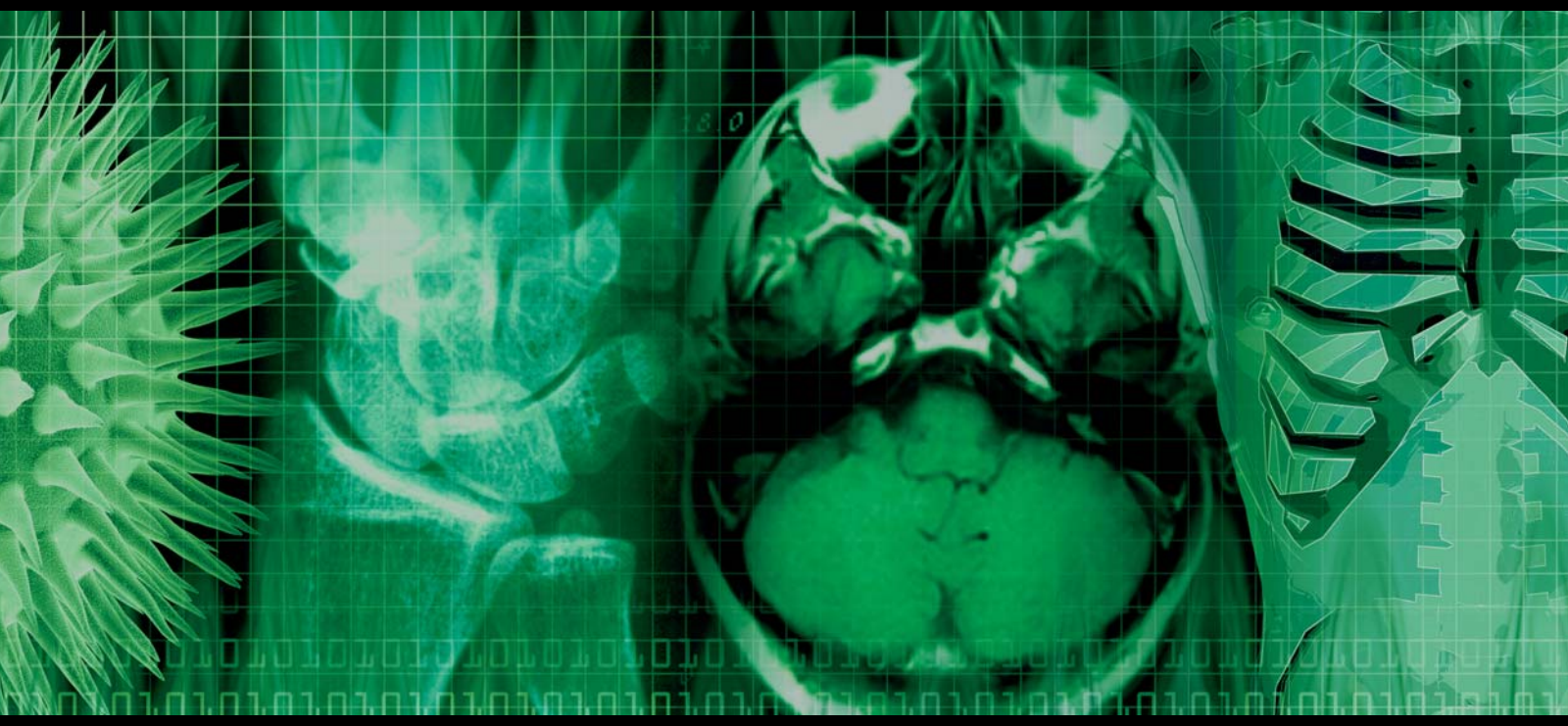
International Journal of Biomedical Imaging

Parallel Computation in Medical Imaging Applications

Guest Editors: Yasser M. Kadah, Khaled Z. Abd-Elmoniem, and Aly A. Farag



Parallel Computation in Medical Imaging Applications

International Journal of Biomedical Imaging

Parallel Computation in Medical Imaging Applications

Guest Editors: Yasser M. Kadah, Khaled Z. Abd-Elmoniem, and Aly A. Farag

Editorial Board

Contents

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 840181, 2 pages
doi:10.1155/2011/840181

Editorial

Parallel Computation in Medical Imaging Applications

Yasser M. Kadah,[1] Khaled Z. Abd-Elmoniem,[2] and Aly A. Farag[3]

[1] *Department of Biomedical Engineering, Cairo University, Giza 12613, Egypt*
[2] *Biomedical and Metabolic Imaging Branch, NIDDK, National Institutes of Health, Bethesda, MD 20892-2560, USA*
[3] *Department of Electrical and Computer Engineering, University of Louisville, Louisville, KY 40292, USA*

Correspondence should be addressed to Yasser M. Kadah, ymk@k-space.org

Received 21 November 2011; Accepted 23 November 2011

There is currently a rapidly growing interest in parallel computation application in various medical imaging and image processing fields. This trend is expected to continue growing as more sophisticated and challenging medical imaging, image processing, and high-order data visualization problems are being addressed. The ongoing cost drop in computational tools and their wide accessibility play a center role as well. Given its short history, this area is still not a well-defined scientific discipline. The selected topics and papers for this special issue shed more light on various aspects of this expanding field and its potential in accelerating medical imaging applications.

This special issue contains eleven papers covering various imaging modalities including MRI, CT, X-ray, US, and optical tomography. The papers demonstrated the potential of parallel computation in medical imaging and visualization in a wide range of applications including image reconstruction, image denoising, motion estimation, deformable registration, diffeomorphic mapping, and modeling.

In the paper entitled "*CUDA-accelerated geodesic ray-tracing for fiber tracking,*" E. van Aart et al. present an accelerated algorithm for brain fiber tracking. Noninvasive diffusion weighted imaging followed by reconstructing the brain fiber structure provides a unique way to inspect the complex structures inside the brain in a microscopic level. However, these processes are computationally expensive. The proposed algorithm utilizes the parallel structure of a graphics processing unit in combination with the CUDA platform to substantially accelerate the execution time of the fiber tracking by a factor up to 40 times compared to a multithreaded CPU implementation.

In the paper entitled "*Efficient probabilistic and geometric anatomical mapping using particle mesh approximation on GPUs,*" L. Ha et al. developed a new three-dimensional deformable registration algorithm for mapping brain datasets. The problem typically involves significant amount of computation time and thus became infeasible for practical purposes. The proposed registration method generates a mapping between anatomies represented as a multi-compartment model. The implementation of the algorithm using particle mesh approximation on graphical processing units (GPUs) achieves the speed up of three orders of magnitudes compared to a CPU reference implementation, making it possible to use the technique in time-critical applications.

In the paper entitled "*Heterogeneous computing for vertebra detection and segmentation in X-ray images,*" F. Lecron et al. address the low computational efficiency of the conventional active shape model (ACM) algorithm and exploit the potential acceleration achieved when ACM is implemented on a parallel computation architecture. The paper demonstrates a global speedup ranging from 3 to 22, in comparison with the CPU implementation.

In the paper entitled "*Mapping iterative medical imaging algorithm on cell accelerator,*" M. Xu and P. Thulasiraman investigate the potential of parallel computation in accelerating the image algebraic reconstruction techniques which in one application may benefit image reconstruction on CT machines. The authors efficiently map the optimized algorithm on the cell broadband engine (BE) for improved performance over CPU version. The implementation on a cell BE is shown to be five times faster when compared to the performance on Sun Fire x4600, a shared memory machine.

In the paper entitled "*GPU-accelerated finite element method for modelling light transport in diffuse optical tomography,*" M. Schweiger introduces a GPU-accelerated finite element solver for the computation of light transport in scattering media. Solutions are presented for both time-domain and frequency-domain problems. A comparison with a CPU-based implementation shows significant performance gains of the graphics-accelerated solution, with improvements of approximately a factor of 10 and 20 for double- and single-precision computations, respectively.

In the area of MRI reconstruction, the paper entitled "*High-performance 3D compressive sensing MRI reconstruction using many-core architectures,*" by D. Kim et al., investigates how different throughput-oriented architectures can benefit compressed sensing (CS) MRI reconstruction algorithm and what levels of acceleration are feasible on different modern platforms. The authors demonstrate that a CUDA-based code running on a GPU can reconstruct a $256 \times 160 \times 80$ volume from an 8-channel acquisition in as fast as 12 seconds, which is a significant improvement over the state of the art. This achievement may potentially bring CS methods even closer to clinical viability.

In the paper entitled "*True 4D image denoising on the GPU,*" A. Eklund et al. show the implementation of a four-dimensional denoising algorithm on a GPU. The algorithm was applied to a 4D CT heart dataset of the resolution $512 \times 512 \times 445 \times 20$. The result is that the GPU can complete the denoising in as fast as 8 minutes. On the contrary, the CPU implementation requires about 50 minutes. The short processing time increases the clinical value of true 4D image denoising significantly.

In the field of simulation and phantom modeling, the paper entitled "*Patient specific dosimetry phantoms using multichannel LDDMM of the whole body,*" by D. J. Tward et al., describes an accelerated automated procedure for creating detailed patient specific pediatric dosimetry phantoms from a small set of segmented organs in a child's CT scan. The algorithm involves full body mappings from adult template to pediatric images using multichannel large deformation diffeomorphic metric mapping with a parallel implementation. The performance of the algorithm was validated on a set of 24 male and 18 female pediatric patients. Running times for the various patients examined ranged from over 30 hours on a single processor to under 1 hour on 24 processors in parallel.

In the paper entitled "*Numerical solution of diffusion models in biomedical imaging on multicore processors,*" L. D'Amore et al. address the solution of nonlinear partial differential equations (PDEs) of diffusion/advection type, underlying most problems in image analysis. As a case study, the paper addresses the segmentation of medical structures and performs a comparative study of numerical algorithms arising from using the semi-implicit and the fully implicit discretization schemes. Comparison criteria take into account both the accuracy and the efficiency of the algorithms including convergence, execution time, and parallel efficiency. This analysis is carried out in a multicore-based parallel computing environment.

In the paper entitled "*On the usage of GPUs for efficient motion estimation in medical image sequences,*" J. Thiyagalingam et al. investigate the mapping of an enhanced motion estimation algorithm to novel GPU architectures. Using a database of three-dimensional ultrasound image sequences, the authors show that the mapping leads to substantial performance gains, up to a factor of 60 and can provide near-real-time performance. The paper also shows how architectural peculiarities of these devices can be best exploited in the benefit of algorithms, most specifically for addressing the challenges related to their access patterns and different memory configurations. The paper further evaluates the performance of the algorithm on three different GPU architectures and performs a comprehensive analysis of the results.

In the paper entitled "*Fast random permutation tests enable objective evaluation of methods for single subject fMRI analysis*" by A. Eklund et al., it is shown that how the computational power of cost-efficient GPUs can be used to speed up random permutation tests. These tests are commonly involved in fMRI data analysis for identifying areas in the brain that are active. However, the computational burden with processing times ranging from hours to days has made them impractical for routine use in single-subject fMRI analysis. A test on GPU with 10000 permutations takes less than a minute, making statistical analysis of advanced detection methods in fMRI practically feasible. To exemplify the permutation-based approach, brain activity maps generated by the general linear model (GLM) and canonical correlation analysis (CCA) are compared at the same significance level.

Acknowledgments

We would like to thank the authors for their excellent contributions to this issue. Many thanks are due to all the reviewers. Without their constructive comments and timely responses, this issue could not have come out.

Yasser M. Kadah
Khaled Z. Abd-Elmoniem
Aly A. Farag

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 627947, 15 pages
doi:10.1155/2011/627947

Research Article

Fast Random Permutation Tests Enable Objective Evaluation of Methods for Single-Subject fMRI Analysis

Anders Eklund,[1,2] Mats Andersson,[1,2] and Hans Knutsson[1,2]

[1] *Division of Medical Informatics, Department of Biomedical Engineering, Linköping University, Linköping, Sweden*
[2] *Center for Medical Image Science and Visualization (CMIV), Linköping University, Linköping, Sweden*

Correspondence should be addressed to Anders Eklund, anders.eklund@liu.se

Received 19 April 2011; Accepted 14 July 2011

Academic Editor: Yasser M. Kadah

Parametric statistical methods, such as *Z*-, *t*-, and *F*-values, are traditionally employed in functional magnetic resonance imaging (fMRI) for identifying areas in the brain that are active with a certain degree of statistical significance. These parametric methods, however, have two major drawbacks. First, it is assumed that the observed data are Gaussian distributed and independent; assumptions that generally are not valid for fMRI data. Second, the statistical test distribution can be derived theoretically only for very simple linear detection statistics. With nonparametric statistical methods, the two limitations described above can be overcome. The major drawback of non-parametric methods is the computational burden with processing times ranging from hours to days, which so far have made them impractical for routine use in single-subject fMRI analysis. In this work, it is shown how the computational power of cost-efficient graphics processing units (GPUs) can be used to speed up random permutation tests. A test with 10000 permutations takes less than a minute, making statistical analysis of advanced detection methods in fMRI practically feasible. To exemplify the permutation-based approach, brain activity maps generated by the general linear model (GLM) and canonical correlation analysis (CCA) are compared at the same significance level.

1. Introduction

Functional magnetic resonance imaging (fMRI) is used in neuroscience and clinic for investigating brain activity patterns and for planning brain surgery. Activity is detected by fitting an activity model to each observed fMRI voxel time series and then testing whether the null hypothesis of no activity can be rejected or not based on the model parameters. Specifically, this test is performed by subjecting a test statistic calculated from the model parameters to a threshold. To control the randomness due to noise in this test procedure, it is desirable to find the statistical significance associated with the detection threshold, that is, how likely it is that a voxel is declared active by chance. When the statistical distribution of the data is known *and* when the probability (null)distribution of the test statistic can be derived, parametric statistics can be used to this end. This is for example the case for the commonly used general linear model (GLM), for which the well-known *t*-test and *F*-test can be derived when the input data are independently Gaussian distributed. However, when the data distribution is not known or the distribution of the test statistics cannot be derived, parametric statistical tests can only yield approximate thresholds or cannot be applied at all. This is generally the case in fMRI analysis as the noise in fMRI data is not Gaussian and independent [1–5]. Even though the noise can be made uncorrelated through a whitening procedure [6, 7], the noise structure must first be estimated using methods that themselves are susceptible to random errors. Accurately accounting for this variance in the test statistic distribution is difficult. Furthermore, more advanced detection approaches often adaptively utilize the spatial context of fMRI activation patterns to improve the detection, or they perform other operations that make the derivation of the test statistic distribution mathematically intractable [8–14]. Said otherwise, only for the very simplest test statistics, such as the GLM, can a parametric test distribution be derived theoretically. On top of the problems described above, the multiple testing problem [15] must be solved because one is generally interested to test whether

there is any activity in the entire brain at all and not just if there is activity in a single voxel. This complicates the derivation of the test statistic distribution even further. To conclude, the parametric statistical approach is applicable only to a very limited set of tests and is subject to many sources of error.

In contrast to the parametric approach, a nonparametric approach does not assume the statistical properties of the input data to be known [16]. Furthermore, there is no need to derive the theoretical distribution of the test statistic, and even thresholds corrected for multiple testing are straightforwardly found. Nonparametric approaches have been studied extensively in functional neuroimaging [10, 17–28]. Semiparametric approaches have also been proposed [29]. In particular, so-called resampling or permutation methods have been studied, which randomly permute or reshuffle the original fMRI data to remove any activation signal but otherwise keep its statistical structure. Thus, thousands of simulated null data sets without activation can be generated and analysed to empirically simulate the null distribution of the test statistic. The major drawback of nonparametric statistical approaches for single-subject fMRI analysis is the computational complexity, requiring hours or days of processing time on regular computer hardware.

Graphics processing units (GPUs) have seen a tremendous development during the last decade and have been applied to a variety of fields to achieve significant speedups, compared to optimized CPU implementations. The main difference between the GPU and the CPU is that the GPU does all the calculations in parallel, while the CPU normally does them in serial. In the field of neuroimaging and neuroscience the use of the GPU is quite new. As single-subject fMRI analysis normally is done separately for each time series in the fMRI data, it suits perfectly for parallel implementations. In our recent work [30] we therefore describe how to preprocess (i.e., apply slice timing correction, motion correction, smoothing, and detrending) and how to statistically analyze the fMRI data on the GPU. The result is a significantly faster analysis than if the CPU is used. For a small fMRI dataset (80 volumes of the resolution $64 \times 64 \times 22$ voxels) all the preprocessing is done in about 0.5 s and the statistical analysis is done under 0.5 ms. Recently, GPUs have also been used to speed up functional connectivity analysis of fMRI data [31, 32]. A final example is the work by Ferreira da Silva [33] that used the GPU to speed up the simulation of a Bayesian multilevel model for fMRI analysis.

In this work, it is shown how nonparametric statistics can be made practical for single-subject fMRI analysis by using the parallel processing power of the GPU. The idea of using the GPU for random permutation tests is not new; it has recently been done in biostatistics [34, 35]. The GPU makes it possible to estimate the null distribution of a test statistic, corrected for multiple testing, in the order of minutes. This has significant implications on the way fMRI analysis can be carried out as it opens the possibility to routinely apply more powerful detection methods than the GLM. As an example, the results of the standard GLM detection is in this work compared with a restricted canonical correlation analysis (CCA) method [9] that adaptively incorporates spatial context in the detection. The short processing time also facilitates deeper investigations into the influence of various noise and detrending models on the statistical significance, as well as validation of approximate parametric approaches such as Bonferroni and random field theory (RFT) [36–39].

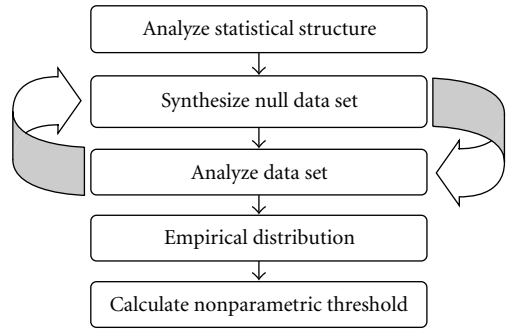


Figure 1: Flowchart containing the main building blocks for nonparametric analysis of single-subject fMRI data.

2. Methods

2.1. Basics of Random Permutation Tests. One subset of nonparametric tests is permutation tests where the statistical analysis is done for all the possible permutations of the data. Complete permutation tests are; however, not feasible if the number of possible permutations is very large. For a time series with 80 samples, there exists $7.16 \cdot 10^{118}$ possible permutations. It is therefore common to instead do *random* permutation tests [40], also called Monte Carlo permutation tests, where the statistical analysis is made for a sufficiently large number of random permutations, for example 10 000, of the data. The main idea is to estimate the null distribution of the test statistics, by generating and analysing surrogate data that is similar to the original data. The surrogate data is generated by permuting, or reshuffling, the data between the different groups to be compared. The main idea of the nonparametric approach is given in Figure 1.

2.2. The Problem of Multiple Testing. By applying a threshold to the activity map, each voxel can be classified as active or inactive. The threshold is normally selected as a level of significance, one may for example want that only voxels that with at least 95% significance are to be considered as active. If a statistical test is repeated and a family-wise error rate α is desired, the error rate for each test must be smaller than α. This is known as the problem of multiple testing. If Bonferroni correction is used, the error rate for each comparison becomes α/N_v, where N_v is the number of tests (voxels). This is a correct solution if the tests are independent. In fMRI it is common to perform the statistical analysis for more than 20 000 brain voxels; if a threshold of $P = 0.05$ is used to consider the voxel as active, the P value becomes 0.05/20000 with Bonferroni correction. The assumptions that are made about the behaviour of the tail of the distribution are thereby critical.

There are three problems with Bonferroni correction in fMRI. First, the test statistics is assumed to, under the null hypothesis, follow a certain distribution, such as Student's t-distribution. Second, the smoothness of the data is not taken into account as the Bonferroni threshold only considers the number of tests. The smoothing increases the spatial correlation of the data and thereby reduces the effective number of independent tests. Third, it is assumed that all the voxels have the same null distribution. To avoid Bonferroni correction, another approach based on Gaussian random field theory [36, 38, 39] has been developed and is used in the statistical parametric mapping (SPM) software (http://www.fil.ion.ucl.ac.uk/spm/). While this approach takes the smoothness of the data into account, several assumptions are necessary for the theory to be valid and it is still assumed that all the voxels have the same null distribution.

The nonparametric approach can be used to solve the problem of multiple testing as well. This is done by estimating the null distribution of the *maximum* test statistic [19, 21, 24, 39] by only saving the maximum test value from each permutation, to get a *corrected* threshold. This means that about 10 000 permutations have to be used [19, 21], while as little as 10 permutations can be enough if an *uncorrected* threshold is sufficient [18, 20, 22].

2.3. Preprocessing of fMRI Time Series. As fMRI time series are temporally correlated [6, 36, 41], the time series have to be preprocessed before they are permuted. Otherwise the exchangeability criterion is not satisfied and the temporal structure is destroyed. Most of the temporal correlations originate from different kinds of trends. In this work these trends are removed by a cubic detrending, such that the mean and any polynomial trend up to the third order is removed, but more advanced detrending is possible [42].

Several approaches have been proposed for the random resampling, the most common being whitening transforms [6, 18, 19], wavelet transforms [22, 26], and Fourier transforms [43]. A comparison of these approaches [27] indicates that whitening performs best, at least for fMRI data that is collected during block-based stimuli paradigms. The whitening transform is done by estimating an autoregressive (AR) model for each time series. This can, for example, be done by solving the equation system that is given by the Yule-Walker equations.

To accurately estimate AR parameters from a small number of time points (80 in our case) is quite difficult. To improve the estimates a spatial Gaussian lowpass filter (8 mm FWHM) is therefore applied to the estimated AR parameters [7]. In statistics this technique is normally known as variance pooling. The optimal amount of smoothing was found by testing the AR estimation procedure on temporally correlated Gaussian noise where the spatial patterns of the AR parameters were known. Our amount of smoothing (8 mm) is less than the first application of smoothing of the parameters (15 mm) [7] but close to the optimal amount of smoothing (6.5–7.5 mm) found by further investigation [44]. It has also been reported that the AR estimates are better without the spatial smoothing [45].

Normalized convolution [46] is used to prevent that the smoothing includes AR parameters from outside the brain. With normalized convolution it is possible to use a certainty value for each sample in the convolution. The certainty weighted filter response cwr is calculated as

$$\text{cwr} = \frac{(c \cdot s) * f}{c * f}, \tag{1}$$

where c is the certainty, s is the signal, f is the filter, $\cdot$ denotes point-wise multiplication, and $*$ denotes convolution. In our case the certainty is set to 1 for the brain voxels and 0 otherwise. Without the normalized convolution the estimated AR parameters at the edge of the brain are too low, as the AR parameters outside the brain are very close to 0. To further improve the estimates, the whitening procedure is iterated 3 times and the AR estimates are accumulated [7, 22] (a higher number of iterations seem to impair the estimates).

To investigate if the time series really are white noise after the whitening, several tests can be applied. One example is the Durbin-Watson test that previously has been used to test if the residuals from the GLM contain autocorrelation [2]. The problem with this test is, however, that it only tests if there is an AR(1) correlation or not, it cannot handle higher-order correlations. A more general test is the Box-Pierce test that tests if at least one of the autocorrelations up to a defined time lag h is significantly different from zero. The Box-Pierce test has also been used for testing whiteness of fMRI data [22]. The Ljung-Box test [47] has been proven to be better than the Box-Pierce test for small sample sizes and is therefore used in our case. The test statistic Q is calculated as

$$Q = N_t(N_t + 2)\sum_{k=1}^{h} \frac{(r_Y(k))^2}{N_t - k}, \tag{2}$$

where N_t is the number of time samples, $r_Y(k)$ is the sample autocorrelation at time lag k, and h is the number of time lags being tested. The test statistic is asymptotically chi-square distributed with $h - p$ degrees of freedom, where p is the order of the AR model used for the whitening, when N_t grows towards infinity. Since our whitening is done with the smoothed AR parameters, the Ljung-Box test is applied to smoothed auto correlations.

Since the spatial correlation should be maintained, but not the temporal, the same permutation is applied to all the time series [19, 43]. When the time series have been permuted, an inverse whitening transform is applied by simulating an AR model, using the permuted whitened time series as innovations.

2.4. Statistical Analysis, GLM and t-Test. The general linear model (GLM) is the most used approach for statistical analysis of fMRI data [37]. For each voxel time series, a linear model is fitted according to

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{3}$$

where $\mathbf{Y}$ is the time series, $\mathbf{X}$ the regressors, $\boldsymbol{\beta}$ the parameters to estimate, and $\boldsymbol{\epsilon}$ the errors. The regressors were created by

convolving the stimulus paradigm with the hemodynamic response function (HRF) (difference of gammas) and its temporal derivative [6]. The two regressors were mean corrected, Euclidean normalized, and orthogonalized. No other regressors were used in the design matrix. The regression weights are estimated with ordinary least squares

$$\hat{\boldsymbol{\beta}} = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{Y}, \tag{4}$$

and the t-test value is then calculated as

$$t = \frac{\mathbf{c}^T\hat{\boldsymbol{\beta}}}{\sqrt{\mathrm{var}(\hat{\boldsymbol{\epsilon}})\mathbf{c}^T(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{c}}}, \tag{5}$$

where $\mathbf{c}$ is the contrast vector.

Prior to the GLM the time series were whitened by using the same AR(1) model for all the voxels [6, 18]. No additional high-pass or low-pass filtering was used. The whitening step prior to the GLM is not necessary for the permutation-based analysis. The purpose of the whitening is to make sure that the errors are temporally uncorrelated, otherwise the assumptions that are necessary for the GLM to generate a true t-value are violated. Without the whitening a true t-value is not obtained, but a pseudo t-value. This is not a problem for the permutation-based analysis as the null distribution of the test statistics is estimated. If the thresholds from random field theory and a random permutation test are to be compared, the whitening has to be done in each permutation.

2.5. Statistical Analysis, CCA. One statistical approach for fMRI analysis that provides more adaptivity to the data is canonical correlation analysis (CCA) [48]. While the GLM works with *one* multidimensional variable (e.g., temporal basis functions, [37]), CCA works with *two* multidimensional variables (e.g., temporal and spatial basis functions, [9]). Ordinary correlation between two one-dimensional variables x and y with zero mean can be written as

$$\rho = \mathrm{Corr}(x, y) = \frac{E[xy]}{\sqrt{E[x^2]\ \mathrm{E}[y^2]}}. \tag{6}$$

This expression can easily be extended to multidimensional variables. The GLM calculates the correlation between one multidimensional variable $\mathbf{x}$ and one one-dimensional variable y according to

$$\rho = \mathrm{Corr}\left(\boldsymbol{\beta}^T\mathbf{x}, y\right), \tag{7}$$

where $\boldsymbol{\beta}$ is the weight vector that determines the linear combination of $\mathbf{x}$. Canonical correlation analysis is a further generalization of the GLM, such that both the variables are multidimensional. The canonical correlation is defined as

$$\rho = \mathrm{Corr}\left(\boldsymbol{\beta}^{\mathrm{T}}\mathbf{x}, \boldsymbol{\gamma}^{\mathrm{T}}\mathbf{y}\right) = \frac{\boldsymbol{\beta}^T\mathbf{C}_{\mathbf{xy}}\boldsymbol{\gamma}}{\sqrt{\boldsymbol{\beta}^T\mathbf{C}_{\mathbf{xx}}\,\boldsymbol{\beta}\ \boldsymbol{\gamma}^T\mathbf{C}_{\mathbf{yy}}\boldsymbol{\gamma}}}, \tag{8}$$

where $\mathbf{C}_{\mathbf{xy}}$ is the covariance matrix between $\mathbf{x}$ and $\mathbf{y}$, $\mathbf{C}_{\mathbf{xx}}$ is the covariance matrix for $\mathbf{x}$, and $\mathbf{C}_{\mathbf{yy}}$ is the covariance matrix for $\mathbf{y}$. The temporal and spatial weight vectors, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, that give the highest correlation are calculated as the eigenvectors of two eigenvalue problems. The canonical correlation is the square root of the corresponding eigenvalue.

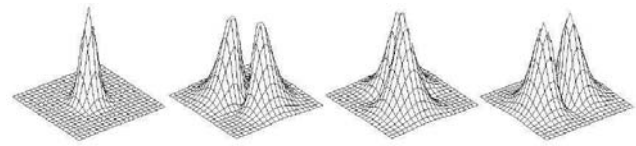


Figure 2: The four smoothing filters that are used for 2D CCA, one small isotropic separable filter and three anisotropic nonseparable filters. For visualization purposes, these filters are interpolated to a subpixel grid.

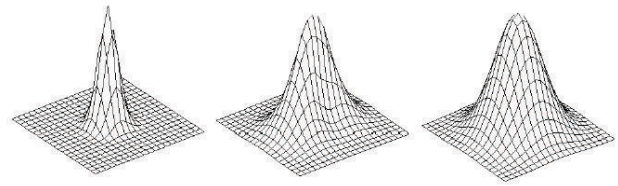


Figure 3: **Left**: A small isotropic lowpass filter can be used by CCA by setting the weights of all the other filters to zero. **Middle**: anisotropic lowpass filters with arbitrary orientation can be created by CCA by first combining the anisotropic filters and then adding the small lowpass filter. **Right**: by using the same weight for all the filters, a large isotropic lowpass filter can be obtained.

The temporal basis functions for CCA are the same as for the GLM. The spatial basis functions can, for example, be neighbouring pixels [8, 10, 49] or a number of anisotropic filters [9] that linearly can be combined to a lowpass filter with arbitrary orientation, to prevent unnecessary smoothing. In contrast to the GLM, an adaptive anisotropic smoothing is obtained, instead of a fix isotropic smoothing. The four smoothing filters that are used for our implementation of 2D CCA are given in Figure 2. Three filters that can be constructed as a linear combination of the four filters are given in Figure 3.

Compared to other approaches that adaptively include spatial information [11–14], the advantage with CCA is that there exists an analytical solution that gives the best weight vectors, while the other approaches have to search for the best combination.

One disadvantage with CCA is that it is difficult to calculate the threshold for a certain significance level, as the distribution of the canonical correlation coefficients is rather complicated. If $\mathbf{x}$ and $\mathbf{y}$ are Gaussian distributed and independent, the joint probability distribution for *all* the sample canonical correlation coefficients is given by [50]

$$f = \prod_{i=1}^{m}\left((r_i^2)^{(n-m-1)/2}(1-r_i^2)^{(N-n-m-1)/2}\prod_{j=i+1}^{m}\left(r_i^2 - r_j^2\right)\right), \tag{9}$$

where N is the number of (time) samples, n and m are the dimensions of the multidimensional variables $\mathbf{x}$ and $\mathbf{y}$, and r_i are the canonical correlation coefficients.

Another problem is that restricted CCA (RCCA) [51] normally is used instead of ordinary CCA, in order to guarantee that the resulting combinations of the temporal and spatial basis functions are plausible. To our knowledge there is no theoretical distribution for restricted canonical correlation coefficients. The only solution to get a significance threshold for RCCA is thus to use nonparametric approaches.

As a 2D version of CCA already had been implemented [30], it was easy to extend the random permutation tests to include CCA as well. The problem with the original 3D CCA approach [9] is that it uses a total of seven 3D filters, and thereby a 7×7 matrix must be inverted for each time series. Our GPU implementation, however, only supports inverting 4×4 matrices, and thereby a maximum of 4 filters. Another approach [52] that uses two 3D filters, one isotropic Gaussian kernel and its derivative (with respect to the width parameter sigma), was therefore used. This makes it possible for CCA to create filters with different sizes, such that the amount of smoothing varies between the voxels. All the resulting filters are, however, isotropic, which makes this version of 3D CCA less adaptive.

2.6. Spatial Smoothing. The smoothing of the fMRI volumes has to be applied in each permutation. If the data is smoothed prior to the whitening transform, the estimated AR parameters will change with the amount of smoothing applied since the temporal correlations are altered by the smoothing. For our implementation of 2D CCA, 4 different smoothing filters are applied. If the smoothing is done prior to the permutations, 4 time series have to be permuted for each voxel and these time series will have different AR parameters. The smoothing will also change the null distribution of each voxel. This is incorrect as the surrogate null data that is created always should have the same properties, regardless of the amount of smoothing that is used for the analysis. If the data is smoothed after the whitening transform, but before the permutation and the inverse whitening transform, the time series that are given by simulating the AR model are incorrect since the properties of the noise are altered. The only solution to this is to apply the smoothing *after* the permutation and the inverse whitening transform, that is, in each permutation. This is also more natural in the sense that the surrogate data first is created and then analysed.

Similarly, if the activity map is calculated as how important each voxel is for a classifier [11–14], the classifier has to be trained in each permutation in order to estimate the null distribution.

2.7. The Complete Algorithm. The complete algorithm can be summarized as follows. The reason why the detrending is done separately, compared to having the detrending basis functions in the design matrix, is that the detrending has to be done separately for the CCA approach.

The whitening in each permutation is only performed to be able to compare the corrected t-thresholds from the random permutation test to the thresholds from Bonferroni correction and random field theory.

(1) Preprocess the fMRI data, that is, apply slice timing correction, motion correction, smoothing, and cubic detrending. To save time, the statistical analysis is only performed for the brain voxels. A simple thresholding technique is used for the segmentation.

(2) Whiten the detrended time series (GLM only).

(3) Apply the statistical analysis to the preprocessed fMRI data and save the test values. These are the original test values t_{voxel}.

(4) Apply cubic detrending to the motion compensated time series.

(5) Remove the best linear fit between the detrended time series and the temporal basis functions in the design matrix, by ordinary least squares, to create residual data (as the *null* distribution is to be estimated). Estimate AR parameters from the residual time series. Apply a spatial smoothing to improve the estimates of the AR parameters. Apply whitening with the smoothed AR parameters. Repeat the whitening procedure 3 times.

(6) For each permutation,

  (i) apply a random permutation to the whitened time series,

  (ii) generate new fMRI time series by an inverse whitening transform, that is, by simulating an AR model in each voxel with the permuted whitened time series as innovations,

  (iii) smooth all the volumes that were generated by the inverse whitening transform,

  (iv) apply cubic detrending to the smoothed time series,

  (v) whiten the detrended time series (GLM only),

  (vi) apply the statistical analysis,

  (vii) find the maximum test value and save it.

(7) Sort the maximum test values.

(8) The threshold for a desired corrected P value is given by extracting the corresponding test value from the sorted maximum test values. If 10 000 permutations are used, the threshold for corrected $P = 0.05$ is given by the sorted maximum test value at location 9500.

(9) The corrected P value at each voxel is calculated as the number of maximum test values, $t\max_i$, that are greater than or equal to the original test value in the voxel, t_{voxel}, divided by the number of permutations N_p

$$p^c_{\text{voxel}} = \frac{\sum_{i=1}^{N_p} (t\max_i \geq t_{\text{voxel}})}{N_p}. \tag{10}$$

A comparison of the flowcharts for a parametric analysis and a nonparametric analysis is given in Figure 4.

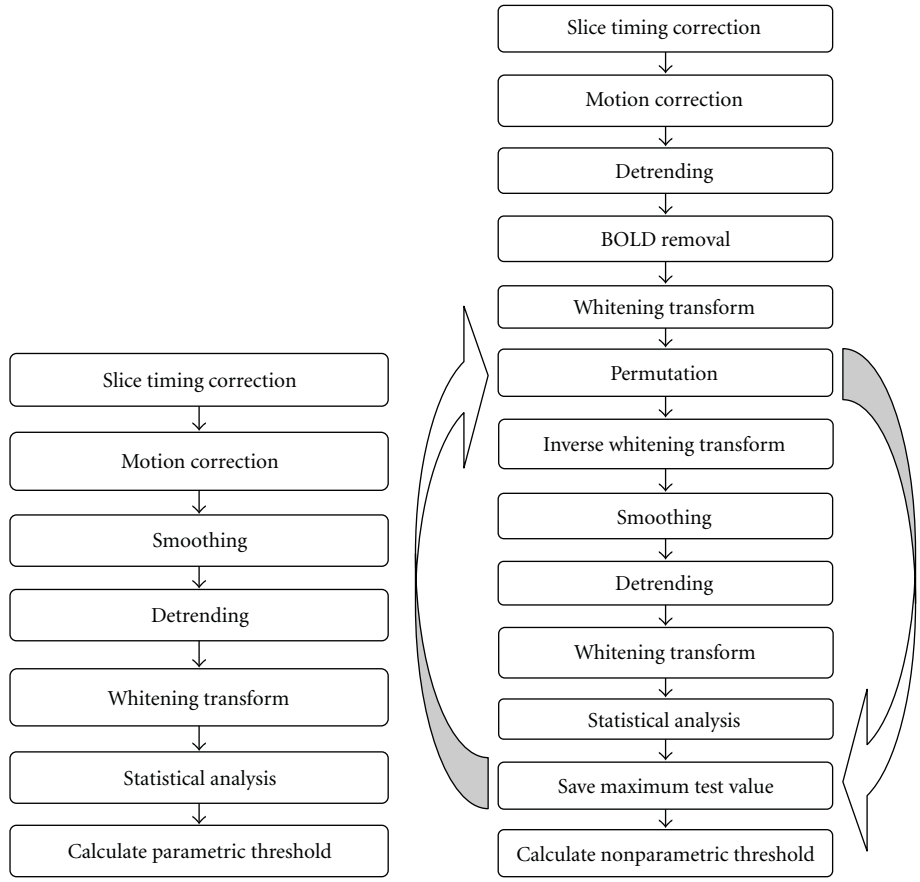


Figure 4: (a) Flowchart for conventional parametric analysis of fMRI data. (b) Flowchart for nonparametric analysis of fMRI data. In each permutation a new null dataset is generated and analysed.

2.8. The Number of Permutations. The number of permutations that are required depends on the desired P value and the accuracy that is required. The standard deviation of the desired (one sided) P value is approximately $\sqrt{p(1-p)/N_p}$, where N_p is the number of permutations [53]. Some examples of desired P value, number of permutations, and relative standard deviation are given in Table 1.

Table 1: Relative standard deviation of the desired P value, as function of desired P value and number of permutations.

Number of Permutations/P value	0.1	0.05	0.01
1000	9.48%	13.78%	31.46%
5 000	4.24%	6.16%	14.07%
10 000	3%	4.35%	9.95%
50 000	1.34%	1.94%	4.45%
100 000	0.95%	1.37%	3.14%

3. GPU Implementation

The random permutation test was implemented with the CUDA (Compute Unified Device Architecture) programming language by Nvidia [54], which is explained by Kirk and Hwu [55]. In this section we will shortly describe how to implement the whitening transform and the random permutation test on the GPU. The interested reader is referred to our recent work [30] for more details and how to implement the other processing steps. The main principle of our GPU implementation is that each GPU thread works on a separate voxel time series.

Our CUDA implementation was compared with a standard *C* implementation and an OpenMP-based implementation. The Open MP (Open Multi-Processing) library lets the user take advantage of all the CPU cores in an easy way. All the implementations have been done in Matlab (Mathworks, Natick, Mass), using the mex interface where *C* and CUDA code can be used together with Matlab. For all implementations, 32 bit floats were used. The used graphics cards were three Nvidia GTX 480, each equipped with 480 processor cores and 1.5 GB of memory, giving a total of 1440 processor cores that run at 1.4 GHz. The used CPU was an Intel Xeon 2.4 GHz with 12 MB of L3 cache and 4 processor cores, and 12 GB of memory was used. The operating system used was Linux Fedora 12 64-bit. The total price of the

computer was about 4000 $, a fraction of the price for a PC cluster with equivalent computational performance.

3.1. Whitening and the Random Permutations. Before the data is permuted an AR model is first estimated for each time series, as previously described. To solve the equation system that is given by the Yule-Walker equations requires a matrix inverse of the size $p \times p$ where p is the order of the AR model. To actually do the matrix inverse in each thread on the GPU is not a problem, even for matrices larger than 4×4, but to do it for a 7×7 matrix requires a lot of float registers and the Nvidia GTX 480 can only use 64 float registers per thread. The CUDA compiler will put the rest of the float variables, that do not fit into the registers, in the local memory which is extremely slow. Due to this it is hard to achieve good performance for matrices that are larger than 4×4. This is also the reason why the original 3D CCA approach, that uses seven 3D filters, cannot be used. Other than this the estimation of the AR parameters suits the GPU well, as the parameters are estimated in exactly the same way for each voxel time series. When the AR parameters have been estimated, they are spatially smoothed in order to improve the estimates. For this purpose a separable 3D convolver, created for 3D CCA and 3D GLM, is used.

For the estimation of AR parameters, the whitening transform, and the inverse whitening transform the shared memory is used to store the p last time points and each GPU thread loops along the time dimension for one voxel.

The permutation step is done by using randomized indices. A permutation matrix of size $N_p \times N_t$ is first generated in Matlab, by using the function *randperm*, and is then copied to the GPU. For each permutation one row of the permutation matrix is copied to the constant memory and is used to read the data in the randomized order. It might seem difficult to achieve coalesced reads when the time samples are to be read in a randomized order, in our case this is however not a problem. The fMRI data is stored as (x, y, z, t) (i.e., x first, then y and so on) and the permutation is only done along the time dimension, and not along the spatial dimensions. Due to this fact it is always possible to read 32 values at the time along x, regardless of the current time point. The code in Algorithm 1 generates a new time series for one voxel, by simulating an AR(4) model using the permuted whitened time series as innovations:

c_Permutation_Vector

is the index vector that contains the random time indices. The inverse whitening transform and the permutation step are thus performed at the same time. The help functions

Get_3D_Index, Get_4D_Index

calculate the linear index for the 3D and the 4D case.

For this kernel, and for most of the other kernels, each thread block contains a total of 512 threads (32 along x, 16 along y, and 1 along z) and uses 16 KB of shared memory (one $16 \times 8 \times 32$ float array) which makes it possible to run three thread blocks in parallel on each multiprocessor. This results in 1536 active threads per multiprocessor and thereby a total of 23 040 active threads on the GPU, which is the maximum for the Nvidia GTX 480.

To find the maximum test value in each permutation, one fMRI slice (64×64 pixels) is first loaded into shared memory. The maximum value of this slice is then found by comparing two values at the time. The number of values is thus first reduced from 4096 to 2048, then to 1024 and after 12 reductions to the maximum test value. The maximum values of the 22 slices are then compared. After each permutation the maximum test value is copied to host memory.

In order to calculate the P value for each voxel, the maximum test values are first copied from host memory to constant memory. Each GPU thread then loops over all the maximum test values and calculates how many of the test values are bigger than or equal to the test value for one voxel.

3.2. Multi-GPU. As our computer contains three graphic cards, a multi-GPU implementation of the analysis was also made, such that each GPU does one-third of the permutations. Each GPU first preprocesses the fMRI data, GPU 1 uses the first part of the permutation matrix, GPU 2 uses the middle part, and GPU 3 uses the last part. The processing time thus scales linearly with the number of GPUs. A short demo of the multi-GPU implementation can be found at http://www.youtube.com/watch?v=wxMqZw0jcOk.

4. Results

In this section we will present the processing times for the different implementations, compare activity maps from GLM and CCA at the same significance level, and compare estimated thresholds from Bonferroni correction, Gaussian random field theory, and random permutation tests.

4.1. Data. Four single-subject datasets have been used to test our algorithms; the test subject was a 50-year-old healthy male. The data was collected with a 1.5 T Philips Achieva MR scanner. The following settings were used: repetition time 2 s, echo time 40 ms, flip angle 90 degrees, and isotropic voxel size 3.75 mm. A field of view of 240 mm thereby resulted in slices with 64×64 pixels, and a total of 22 slices were collected every other second. The experiments were 160 s long, resulting in 80 volumes to be processed. The datasets contain about 20 000 within-brain voxels.

4.1.1. Motor Activity. For the *Motor 1* dataset the subject periodically activated the left hand (20 s activity, 20 s rest), and for the *Motor 2* dataset the subject periodically activated the right hand.

4.1.2. Language Activity. For the *Language* dataset the subject periodically performed a reading task (20 s activity, 20 s rest). The task was to read sentences and determine if they were reasonable or not.

4.1.3. Null. For the *null* dataset the subject simply rested during the whole experiment.

```
float alpha1 = alphas1[Get_3D_Index(x,y,z,DATA_W,DATA_H)];
float alpha2 = alphas2[Get_3D_Index(x,y,z,DATA_W,DATA_H)];
float alpha3 = alphas3[Get_3D_Index(x,y,z,DATA_W,DATA_H)];
float alpha4 = alphas4[Get_3D_Index(x,y,z,DATA_W,DATA_H)];

s_Y[threadIdx.y][0][threadIdx.x] =
whitened_volumes[Get_4D_Index(x,y,z,c_Permutation_Vector[0],DATA_W,DATA_H,DATA_D)];
s_Y[threadIdx.y][1][threadIdx.x] = alpha1 * s_Y[threadIdx.y][0][threadIdx.x]
+ whitened_volumes[Get_4D_Index(x,y,z,c_Permutation_Vector[1],DATA_W,DATA_H,DATA_D)];
s_Y[threadIdx.y][2][threadIdx.x] = alpha1 * s_Y[threadIdx.y][1][threadIdx.x]
+ alpha2 * s_Y[threadIdx.y][0][threadIdx.x]
+ whitened_volumes[Get_4D_Index(x,y,z,c_Permutation_Vector[2],DATA_W,DATA_H,DATA_D)];
s_Y[threadIdx.y][3][threadIdx.x] = alpha1 * s_Y[threadIdx.y][2][threadIdx.x]
+ alpha2 * s_Y[threadIdx.y][1][threadIdx.x] + alpha3 * s_Y[threadIdx.y][0][threadIdx.x]
+ whitened_volumes[Get_4D_Index(x,y,z,c_Permutation_Vector[3],DATA_W,DATA_H,DATA_D)];

permuted_volumes[Get_4D_Index(x,y,z,0,DATA_W,DATA_H,DATA_D)] = s_Y[threadIdx.y][0][threadIdx.x];
permuted_volumes[Get_4D_Index(x,y,z,1,DATA_W,DATA_H,DATA_D)] = s_Y[threadIdx.y][1][threadIdx.x];
permuted_volumes[Get_4D_Index(x,y,z,2,DATA_W,DATA_H,DATA_D)] = s_Y[threadIdx.y][2][threadIdx.x];
permuted_volumes[Get_4D_Index(x,y,z,3,DATA_W,DATA_H,DATA_D)] = s_Y[threadIdx.y][3][threadIdx.x];

// Loop over time points
for (t = 4; t < DATA_T; t++)
{
   s_Y[threadIdx.y][4][threadIdx.x] =
     alpha1 * s_Y[threadIdx.y][3][threadIdx.x]
   + alpha2 * s_Y[threadIdx.y][2][threadIdx.x]
   + alpha3 * s_Y[threadIdx.y][1][threadIdx.x]
   + alpha4 * s_Y[threadIdx.y][0][threadIdx.x]
   + whitened_volumes[Get_4D_Index(x,y,z,c_Permutation_Vector[t],DATA_W,DATA_H,DATA_D)];

   permuted_volumes[Get_4D_Index(x,y,z,t,DATA_W,DATA_H,DATA_D)] = s_Y[threadIdx.y][4][threadIdx.x];

   // Save old values
   s_Y[threadIdx.y][0][threadIdx.x] = s_Y[threadIdx.y][1][threadIdx.x];
   s_Y[threadIdx.y][1][threadIdx.x] = s_Y[threadIdx.y][2][threadIdx.x];
   s_Y[threadIdx.y][2][threadIdx.x] = s_Y[threadIdx.y][3][threadIdx.x];
   s_Y[threadIdx.y][3][threadIdx.x] = s_Y[threadIdx.y][4][threadIdx.x];
}
```

Algorithm 1

4.2. Processing Times. The processing times for the random permutation tests, for the different implementations, are given in Tables 2 and 3. The reason why the processing time does not scale linearly with the number of permutations is that it takes some time to copy the data to and from the GPU. Before the permutations are started, the fMRI data is preprocessed on the GPU and this takes about 0.5 s. The processing times for the different processing steps can be found in our recent work [30].

The reason why the processing time for CCA is much longer than for the GLM for the CPU implementations is that the 2D version of CCA uses one separable filter and three nonseparable filters for the smoothing while the GLM uses one separable filter. For the GPU implementation the 2D smoothing can be done extremely fast by using the shared memory.

4.3. Verifying the Whitening Procedure. To verify that the whitening procedure prior to the permutations works correctly, the Ljung-Box test was applied to each residual time series. The Ljung-Box test was applied to the four datasets after detrending, BOLD removal, and whitening with AR models of different order. The test was applied for 1–10 time lags (i.e., 10 tests), and the mean number of nonwhite voxels was saved. A voxel-wise threshold of $P = 0.05$ was used, that is, $\chi^2_{0.95,h-p}$ where h is the number of time lags tested and p is the order of the AR model used. This means that the test only can be applied to certain time lags, since the degrees of freedom otherwise become zero or negative. The results with spatial smoothing of the auto correlations are given in Figure 5 and the results without spatial smoothing are given in Figure 6. The results for Gaussian white noise are included as reference when no smoothing is applied to the auto correlations. With the spatial smoothing, the number of voxels classified as nonwhite for the Gaussian noise is always zero.

If no smoothing is applied to the estimated auto correlations prior to the Ljung-Box test, the test statistic cannot be trusted as the standard deviation of the estimated auto correlations is too high. The reason why the number of nonwhite voxels increases, when no smoothing is applied to the auto correlations and when the degree of the AR model increases, is that the critical threshold of the Ljung-Box test decreases as the order of the AR model increases.

From the results in Figures 5 and 6 we draw the conclusion that cubic detrending and an individual AR(4) whitening is necessary to whiten the *Motor 1*, *Motor 2,* and *Language* datasets while an individual AR(5) or AR(6) whitening is necessary for the *Null* dataset. Long-term autocorrelations have previously been reported for resting state fMRI data.

For all the datasets an individual AR(4) whitening was therefore used prior to the permutations and in each permutation to generate new null data. For the *null* dataset a higher-order AR model is necessary, but to estimate an

Table 2: Processing times for random permutation tests with the GLM for the different implementations.

Number of permutations with GLM	C	OpenMP	CUDA, 1 × GTX 480	CUDA, 3 × GTX 480
1000	25 min	3.5 min	25.2 s	8.4 s
5 000	2 h 5 min	17.5 min	1 min 42 s	33.9 s
10 000	4 h 10 min	35 min	3 min 18 s	65.8 s
50 000	20 h 50 min	2 h 55 min	16 min 30 s	5 min 30 s
100 000	1 day 17 h 40 min	5 h 50 min	33 min	11 min

Table 3: Processing times for random permutation tests with 2D CCA for the different implementations.

Number of permutations with 2D CCA	C	OpenMP	CUDA, 1 × GTX 480	CUDA, 3 × GTX 480
1000	1 h 40 min	14 min 50 s	22.2 s	7.4 s
5 000	8 h 20 min	1 h 14 min	1 min 24 s	28 s
10 000	16 h 37 min	2 h 28 min	2 min 42 s	54 s
50 000	3 days 11 h	12 h 22 min	13 min 30 s	4 min 30 s
100 000	6 days 22 h	24 h 43 min	27 min	9 min

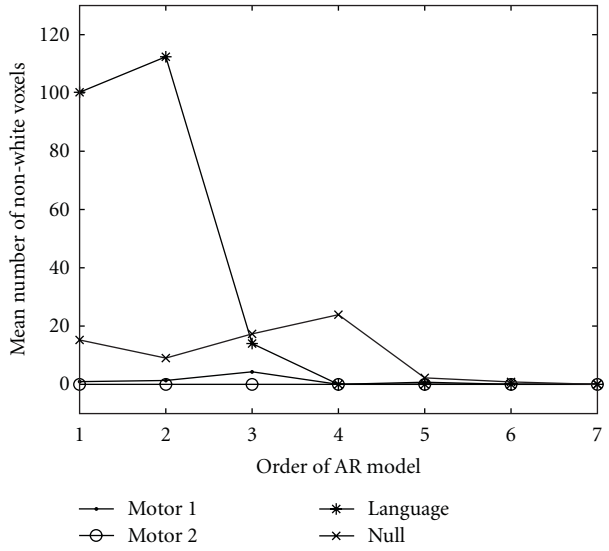


Figure 5: Mean number of voxels classified as nonwhite by the Ljung-Box test (1–10 time lags were tested and the mean number of nonwhite voxels for the 10 tests was saved). Prior to the Ljung-Box test the estimated auto correlations were spatially smoothed. The number of nonwhite voxels for Gaussian white noise is always zero.

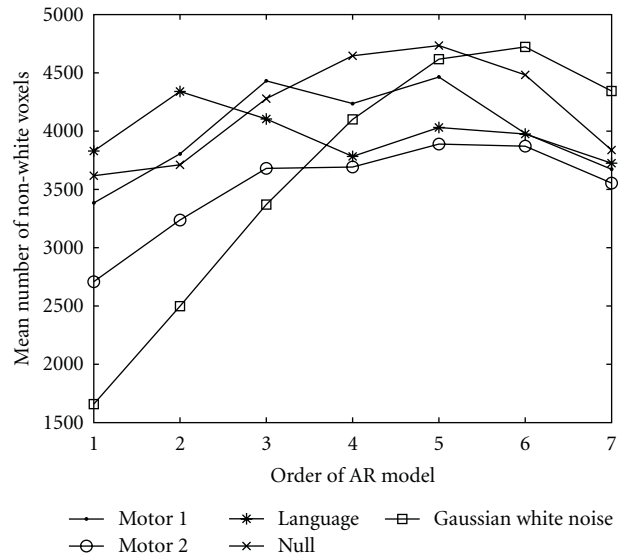


Figure 6: Mean number of voxels classified as nonwhite by the Ljung-Box test (1–10 time lags were tested and the mean number of nonwhite voxels for the 10 tests was saved). No spatial smoothing was applied to the estimated auto correlations prior to the Ljung-Box test. The number of nonwhite voxels for Gaussian white noise is included as reference (no whitening was applied to the noise).

AR(5) model requires matrix inverses of 5 × 5 matrices for each voxel time series, which our GPU implementation does not support. Therefore, the voxels in the *null* dataset that were considered as nonwhite after the AR(4) whitening were instead removed from the random permutation test.

4.4. Verifying the Random Permutation Test. To verify that our random permutation test works correctly, all the preprocessing steps were removed and Gaussian white noise was used as data. The stimulus paradigm convolved with the HRF and its temporal derivative were used as regressors, and a t-test value was calculated for each voxel. A spatial mask from a real fMRI dataset was used to get the same number of brain voxels. A threshold for corrected $P = 0.05$ was calculated, by using 100 000 permutations, and then 10 000 noise datasets were generated (for each amount of smoothing), analysed, and thresholded. If the calculated threshold is correct, 500 of the noise datasets should contain a test value that is higher than the threshold. The family-wise error rate (FWE) was estimated for the thresholds from Bonferroni correction, Gaussian random field theory, and the random permutation test and is given in Figure 7.

4.5. GLM versus CCA. With the random permutation test it is possible to calculate corrected P values for fMRI analysis by CCA, and thereby activity maps from GLM and CCA

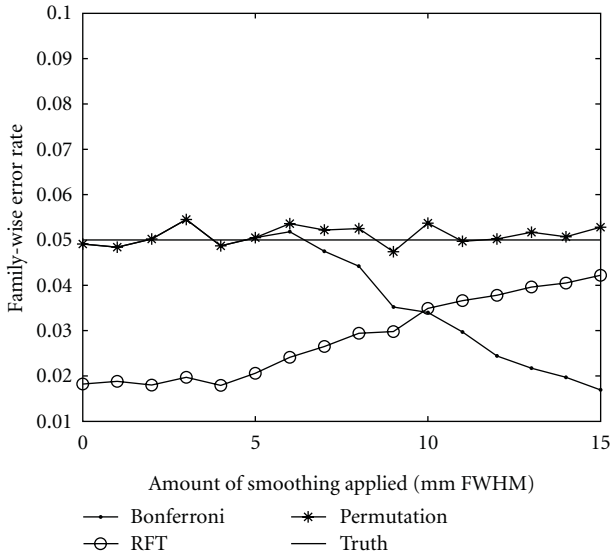


Figure 7: A comparison of family-wise error rates for Gaussian white noise for three different approaches to calculate an activity threshold, corrected for multiple testing.

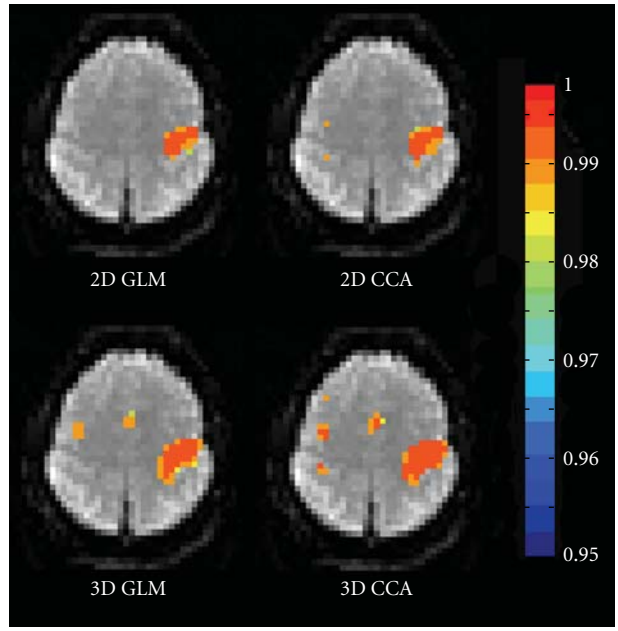


Figure 8: **Top**: A comparison between corrected P values from 2D GLM (left) and 2D CCA (right), calculated from a random permutation test with 10 000 permutations. The activity maps are thresholded at the same significance level (corrected $P = 0.05$). The GLM used one isotropic 8 mm FWHM 2D Gaussian smoothing kernel while CCA used one isotropic 2D Gaussian kernel and 3 anisotropic 2D Gaussian kernels, designed such that the largest possible filter that CCA can create has an FWHM of 8 mm. The neurological display convention is used (left is left), $1-p$ is shown instead of p. Note that CCA detects active voxels in the left motor cortex and in the left somatosensory cortex that are not detected by the GLM. **Bottom**: A comparison between corrected P values from 3D GLM (left) and 3D CCA (right), calculated from a random permutation test with 10 000 permutations. The activity maps are thresholded at the same significance level (corrected $P = 0.05$). The GLM used one isotropic 8 mm FWHM 3D Gaussian smoothing kernel while CCA used one isotropic 3D Gaussian kernel and its derivative, designed such that the largest possible filter that CCA can create has a FWHM of 8 mm. The neurological display convention is used (left is left), $1-p$ is shown instead of p. Note that CCA detects active voxels in the left somatosensory cortex that are not detected by the GLM.

can finally be compared at the same significance level. The activity maps are given in Figure 8. For these comparisons, the *Motor 1* dataset was used, 10 000 permutations were used both for GLM and CCA. The activity maps were thresholded at the same significance level, corrected $P = 0.05$. With 8 mm of 2D smoothing, GLM detects 302 significantly active voxels while CCA detects 344 significantly active voxels. With 8 mm of 3D smoothing, GLM detects 475 significantly active voxels while CCA detects 684 significantly active voxels. The aim of this comparison is not to prove that CCA has a superior detection performance, but to show that objective evaluation of different methods for single-subject fMRI analysis becomes practically possible by using fast random permutation tests.

For RCCA there is no theoretical distribution to calculate a threshold from, and therefore the corrected thresholds for the restricted canonical correlation coefficients are also presented, 10 000 permutations were used to calculate each threshold. Figure 9 shows the found thresholds for 2D CCA and 3D CCA for the *Motor 1* dataset. Similar results were obtained for the other datasets. Since fMRI analysis by CCA results in an adaptive smoothing, compared to a fix smoothing with the GLM, the amount of smoothing varies between the voxels. Due to this, the plots show the corrected thresholds for the different *maximum* amounts of smoothing that can be applied by CCA. These plots would have taken a total of about 14 days to generate with a standard C implementation, with our multi-GPU implementation they took about 30 minutes to generate.

4.6. Comparison of Methods for Calculating Corrected Thresholds. As the null distribution of the maximum t-test statistics can be estimated, it is possible to compare the thresholds that are given by Bonferroni correction, Gaussian random field theory, and a random permutation test (which should give the most correct threshold); 10 000 permutations were used for the random permutation test.

Figure 10 shows the found thresholds for the *Motor 1* dataset, for different amounts of smoothing. Similar results were obtained for the other datasets. To our knowledge, a comparison of thresholds from Bonferroni correction, Gaussian random field theory, and a random permutation test has previously only been done for *multi-subject* fMRI [24]. These plots would have taken a total of about 5.5 days to generate with a standard C implementation; with our multi-GPU implementation they took about 38 minutes to generate.

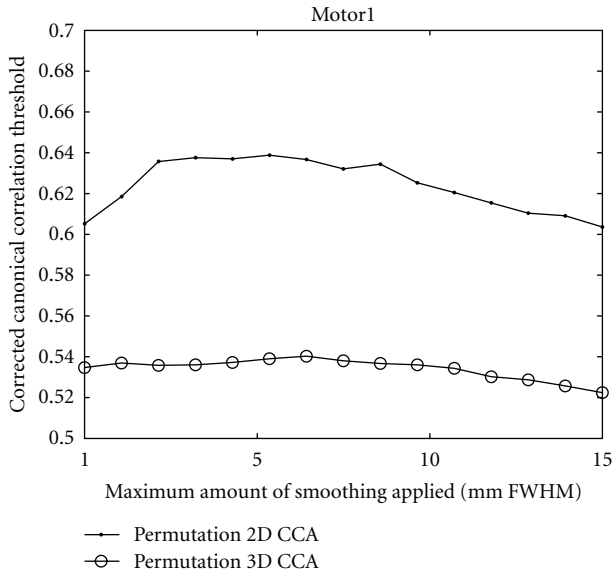


FIGURE 9: Canonical correlation thresholds, for corrected $P = 0.05$, as function of the maximum amount of smoothing that can be applied by CCA. The Motor 1 dataset was used.

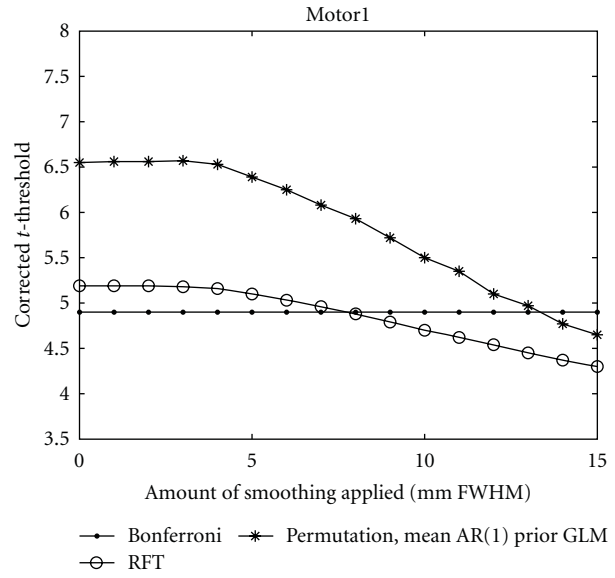


FIGURE 10: A comparison of t-thresholds, for corrected $P = 0.05$, from three approaches to calculate a corrected threshold, as function of the amount of smoothing applied. The Motor 1 dataset was used.

Figure 11 shows the estimated maximum t distribution for the *Motor 1* dataset; 8 mm of smoothing was applied to the volumes in each permutation.

4.7. Distributions of Corrected t-Thresholds. As a final result, distributions of the corrected t-thresholds are presented. The random permutation test for the GLM was repeated 1000 times and the resulting thresholds were saved. The *Motor 1* dataset was used with 8 mm of smoothing. The threshold

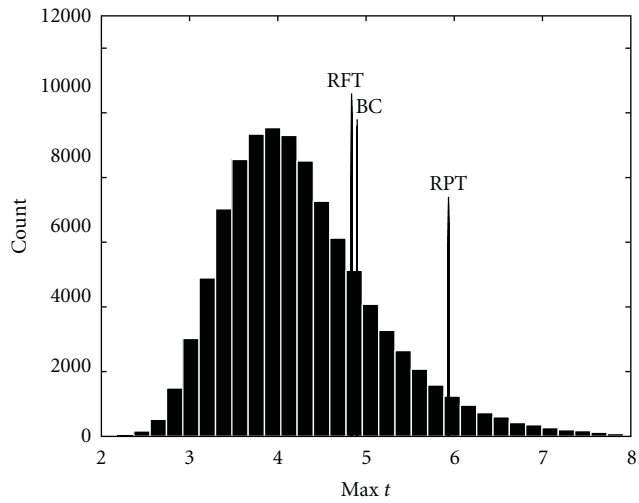


FIGURE 11: The estimated null distribution of the maximum t-test value from the GLM, when 8 mm smoothing was applied. The calculated thresholds for corrected $P = 0.05$ for the three approaches are marked with BC (Bonferroni correction), RFT (random field theory), and RPT (random permutation test).

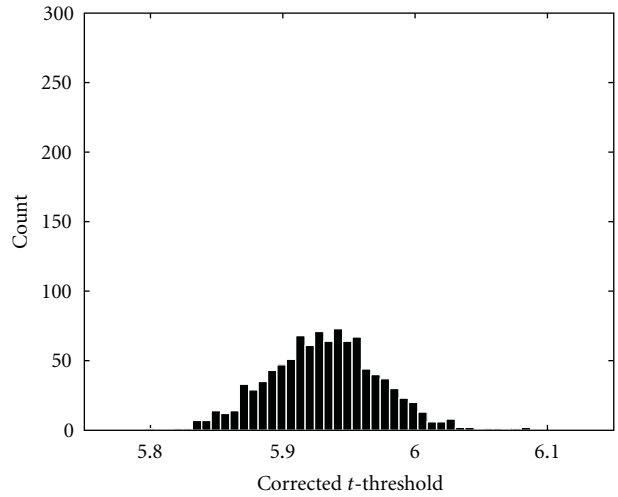


FIGURE 12: The distribution of the corrected t-threshold when 1000 permutations were used.

distribution for 1000 permutations is given in Figure 12 and the threshold distribution for 10 000 permutations is given in Figure 13. The standard deviation of the threshold calculated with 1000 permutations is 0.0364, and the standard deviation of the threshold calculated with 10 000 permutations is 0.0114. According to [53] the standard deviation should decrease with $\sqrt{10}$ if 10 times more permutations are used. The difference in standard deviation between the threshold calculated with 1000 and 10 000 permutations is very close to this approximation.

If 1000 permutations are used (and it is assumed that the estimated distribution is correct), the estimated P value varies between 0.044 and 0.059 if the standard deviation of the corrected threshold is subtracted or added. For 10 000

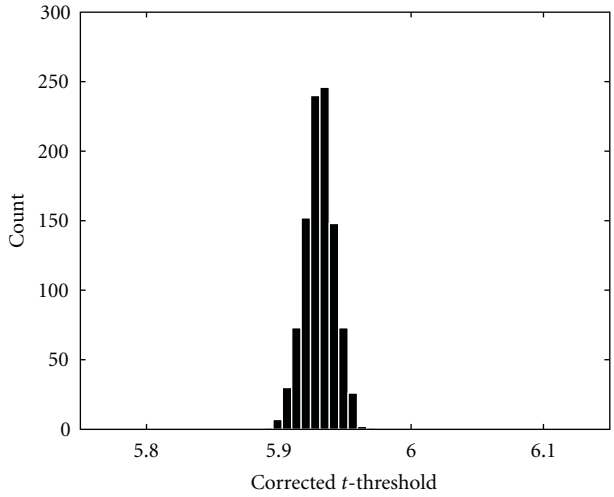


Figure 13: The distribution of the corrected t-threshold when 10 000 permutations were used.

permutations the estimated P value varies between 0.048 and 0.052. This is close to the expected relative standard deviation given in Table 1. It is very important to know the variance of the P values as it tells us how reliable the estimates are.

These plots would have taken a total of about 17.4 and 174 days to generate with a standard C implementation. With the multi-GPU implementation they took about 2.3 and 18 hours to generate.

5. Discussion

With the help of fast random permutation tests it is possible to objectively evaluate activity maps from any test statistics easily, by using the same significance level. As an example of this we compare activity maps from GLM and CCA. It is also possible to investigate how a change in the preprocessing (e.g., the amount of smoothing or the whitening applied) affects the distribution of the maximum test statistics. The search for the best test statistics, that gives the best separation between active and inactive voxels, can now be started. To use simple test statistics and hope that the data is normally distributed and independent is no longer necessary.

5.1. Processing Times. As can be seen in the tables, a lot of time is saved by using the GPU. Most of the time is saved in the smoothing step. The tables clearly show that the GPU, or an advanced PC cluster, is a must for random permutation tests that include smoothing. To do 100 000 permutations with CCA takes about 7 days with the C implementation, about a day with the OpenMP implementation, and about 9 minutes with the multi-GPU implementation. The speedup is about 1100 between the C implementation and the multi-GPU implementation and about 170 between the OpenMP implementation and the multi-GPU implementation.

It should be noted that these processing times are for 80 volumes and 20 000 brain voxels, but it is not uncommon that an fMRI dataset contains 150 volumes and 30 000 brain voxels, which triples the processing times.

The main problem with a long processing time is the software development. In order to test and verify that a program works correctly, the program has to be launched a large number of times. During the development of the routines and the writing of the paper we ran the complete analysis, with 1000–100 000 permutations, at least 3000 times. For the GLM this means that at least 6000 hours of processing time were saved, compared to the C implementation. This is equivalent to 750 working days.

With the power of the GPU it is even possible to look at the distributions of the corrected thresholds that otherwise could take as much as 6 months of processing time to estimate.

The processing time for 10 000 permutations with GLM and smoothing is about 3.5 minutes with one GPU. This is perhaps too long for clinical applications, but we believe that it is fast enough for researchers to use it in their daily work.

5.2. GLM versus CCA. With the help of the GPU it is finally possible to compare activity maps from GLM and CCA at the same significance level. Even if CCA has a superior detection performance compared to the GLM, its use has been limited. One major reason for this is that it is hard to set a (corrected) threshold for CCA.

The presented activity maps show that the CCA approach in general, due to its spatial adaptivity, finds a higher number of significantly active voxels than the GLM approach. With 2D smoothing CCA finds some significantly active voxels in the left motor cortex and in the left somatosensory cortex that are not detected by the GLM. With 3D smoothing CCA finds some significantly active voxels in the left somatosensory cortex that are not detected by the GLM. We have thereby confirmed previous results that fMRI analysis by CCA can result in a higher number of significantly active voxels [9, 10, 56].

It might seem strange that the corrected canonical correlation thresholds do not decrease as rapidly as the corrected t-thresholds when the maximum amount of smoothing increases. By using CCA an adaptive smoothing is obtained, such that the amount of smoothing varies between the voxels. The CCA approach will choose the amount (and orientation) of smoothing that results in the highest canonical correlation, as shown in Figure 3. This is one of the main advantages with CCA, since it, for example, prevents that too much smoothing is used in small activity areas. If the maximum canonical correlation is found by only using the small lowpass filter, the maximum canonical correlation might not change significantly when the maximum amount of smoothing is increased since CCA probably will choose to only use the small lowpass filter once again. The consequence is that it is hard to predict how the maximum test value will change as a function of the maximum amount of smoothing.

The corrected thresholds are lower for 3D CCA than for 2D CCA. This is explained by the fact that the 2D version is adaptive in both scale and orientation, and it can thereby find higher correlations than the 3D version that only is adaptive

in scale. With more advanced GPUs, the original 3D CCA approach, with 7 filters, can be used to obtain more spatial adaptivity in 3D.

5.3. Comparison of Methods for Calculating Corrected Thresholds. The comparison between the thresholds from Bonferroni correction, Gaussian random field theory, and the random permutation test shows some interesting results. The thresholds from the random permutation test are the highest. For the GLM approach to be valid, the data is assumed to be normally distributed as well as independent. For the multiple testing problem, the parametric approaches also assume a common null distribution for each voxel while the permutation approach does not [24]. There are thus, at least, three sources of error for the parametric approaches.

As a t-value is calculated for each time series, the normality condition should be investigated for each time series separately [2], for example, by a Kolmogorov-Smirnov test or a Shapiro-Wilk test. These tests are, however, not very reliable if there are only 80 time points for each voxel. The *maximum* t-distribution is very sensitive to deviations from normality [57], while the standard t-distribution is rather robust. If a few voxel time series have a distribution that deviates from normality, this is sufficient to affect the maximum t-distribution and thereby the threshold [24]. This will be captured by the random permutation test but not by the parametric tests.

The distribution of the t-test values from the *Null* dataset does not strictly follow a Student's t-distribution, especially if 10 mm smoothing is used. The tails are not longer but slightly thicker than a true t-distribution. When a mean AR(1) whitening was used for a conventional analysis of the *Null* dataset, the t-test value that is bigger than 95% of the test values is 1.75 when no smoothing is used, 1.66 when 5 mm of smoothing is used, and 1.59 when 10 mm smoothing is used. The theoretical threshold for uncorrected $P = 0.05$, calculated from the Student's t-distribution, is 1.66. This explains why the thresholds from the random permutation test are higher than the thresholds from Bonferroni correction.

It is commonly assumed that the noise in MRI is normally distributed, but due to the fact that only the magnitude of the MRI data is used, the noise is actually Rician distributed [1]. The original (complex valued) noise in MRI is normally distributed, but the magnitude operation after the inverse Fourier transform in the image reconstruction process is not linear and thereby changes the distribution of the noise. The distribution of the fMRI noise is more complicated as there are several sources of artefacts and the difference between images is used to calculate the test statistics [2–5]. The consequence for fMRI is that the residuals from the GLM might not be normally distributed, even if the model is valid. For the model to be valid, all possible artefacts that can arise have to be modelled. This includes motion-related artefacts, breathing artefacts, pulse artefacts, and MR scanner imperfections. To make a perfect model for all these artefacts is a big challenge on its own.

Another problem for the random field theory approach is that the activity map has to be sufficiently smooth in order to approximate the behaviour of a continuous random field. The smoothness also has to be estimated from the data and it is assumed that it is constant in the brain. These assumptions and several others [24, 39] have to be met in order for the random field theory approach to be valid. For the random permutation test some assumptions also have to be made, for example that the time series are correctly whitened before the permutations. The number of necessary assumptions for the random permutation test is, however, significantly lower than that for the parametric approaches.

5.4. Future Work. In this paper we have only described what is known as a *single-threshold* permutation test, but other types of permutation tests can be more powerful. Examples of this are the so-called *step-down* and *step-up* permutation tests. These permutation tests are even more computationally demanding; it can, for example, be necessary to reestimate the maximum null distribution for each voxel. It is also possible to use the mass of a cluster [28, 58] instead of the voxel intensity, or a combination of the voxel intensity and the cluster extent [25]. The random permutation tests can also be used in order to calculate significance thresholds for functional connectivity analysis [31, 32].

The GPU can of course also be used to speed up permutation tests for multi-subject fMRI and multi-subject PET, and not only for single-subject fMRI. The only drawback with the GPU that has been encountered so far is that some test statistics, like 3D CCA, are harder to implement on the GPU than on the CPU, due to the current limitations of the GPU. It must also be possible to calculate the test statistics in parallel, otherwise the GPU will not provide any speedup.

6. Conclusions

We have presented how to apply random permutation tests for single-subject analysis of fMRI data by using the graphics processing unit (GPU). Our work enables objective evaluation of arbitrary methods for single-subject fMRI analysis. As a pleasant side effect, the problem of multiple testing is solved in a way that significantly reduces the number of necessary assumptions. To our knowledge, our implementation is the first where the smoothing is done *in each permutation*. In previous papers about permutation tests in fMRI, it is neglected that the smoothing has to be done in each permutation for the analysis to be correct.

Acknowledgments

This work was supported the Linnaeus center CADICS, funded by the Swedish research council. The fMRI data was collected at the Center for Medical Image Science and Visualization (CMIV). The authors would like to thank the NovaMedTech project at Linköping University for financial support of our GPU hardware and Johan Wiklund for support with the CUDA installations.

References

[1] H. Gudbjartsson and S. Patz, "The Rician distribution of noisy MRI data," *Magnetic Resonance in Medicine*, vol. 34, no. 6, pp. 910–914, 1995.

[2] W. L. Luo and T. E. Nichols, "Diagnosis and exploration of massively univariate neuroimaging models," *NeuroImage*, vol. 19, no. 3, pp. 1014–1032, 2003.

[3] O. Friman, I. Morocz, and C.-F. Westin, "Examining the whiteness of fMRI noise," in *Proceedings of the ISMRM Annual Meeting*, p. 699, 2005.

[4] T. E. Lund, K. H. Madsen, K. Sidaros, W. L. Luo, and T. E. Nichols, "Non-white noise in fMRI: does modelling have an impact?" *NeuroImage*, vol. 29, no. 1, pp. 54–66, 2006.

[5] A. M. Wink and J. B. T. M. Roerdink, "BOLD noise assumptions in fMRI," *International Journal of Biomedical Imaging*, vol. 2006, Article ID 12014, 2006.

[6] K. J. Friston, O. Josephs, E. Zarahn, A. P. Holmes, S. Rouquette, and J. B. Poline, "To smooth or not to smooth? Bias and efficiency in fMRI time-series analysis," *NeuroImage*, vol. 12, no. 2, pp. 196–208, 2000.

[7] K. J. Worsley, C. H. Liao, J. Aston et al., "A general statistical analysis for fMRI data," *NeuroImage*, vol. 15, no. 1, pp. 1–15, 2002.

[8] O. Friman, J. Cedefamn, P. Lundberg, M. Borga, and H. Knutsson, "Detection of neural activity in functional MRI using canonical correlation analysis," *Magnetic Resonance in Medicine*, vol. 45, no. 2, pp. 323–330, 2001.

[9] O. Friman, M. Borga, P. Lundberg, and H. Knutsson, "Adaptive analysis of fMRI data," *NeuroImage*, vol. 19, no. 3, pp. 837–845, 2003.

[10] R. Nandy and D. Cordes, "A novel nonparametric approach to canonical correlation analysis with applications to low CNR functional MRI data," *Magnetic Resonance in Medicine*, vol. 49, pp. 1152–1162, 2003.

[11] J. Mourão-Miranda, A. L. W. Bokde, C. Born, H. Hampel, and M. Stetter, "Classifying brain states and determining the discriminating activation patterns: support Vector Machine on functional MRI data," *NeuroImage*, vol. 28, no. 4, pp. 980–995, 2005.

[12] N. Kriegeskorte, R. Goebel, and P. Bandettini, "Information-based functional brain mapping," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 103, no. 10, pp. 3863–3868, 2006.

[13] F. D. Martino, G. Valente, N. Staeren, J. Ashburner, R. Goebel, and E. Formisano, "Combining multivariate voxel selection and support vector machines for mapping and classification of fMRI spatial patterns," *NeuroImage*, vol. 43, no. 1, pp. 44–58, 2008.

[14] M. B. Åberg and J. Wessberg, "An evolutionary approach to the identification of informative voxel clusters for brain state discrimination," *IEEE Journal on Selected Topics in Signal Processing*, vol. 2, no. 6, pp. 919–928, 2008.

[15] Y. Hochberg and A. C. Tamhane, *Multiple Comparison Procedures*, John Wiley & Sons, New York, NY, USA, 1987.

[16] S. Siegel, "Nonparametric statistics," *The American Statistician*, vol. 11, pp. 13–19, 1957.

[17] A. P. Holmes, R. C. Blair, J. D. G. Watson, and I. Ford, "Nonparametric analysis of statistic images from functional mapping experiments," *Journal of Cerebral Blood Flow & Metabolism*, vol. 16, no. 1, pp. 7–22, 1996.

[18] E. Bullmore, M. Brammer, S. C. R. Williams et al., "Statistical methods of estimation and inference for functional MR image analysis," *Magnetic Resonance in Medicine*, vol. 35, no. 2, pp. 261–277, 1996.

[19] J. J. Locascio, P. J. Jennings, C. I. Moore, and S. Corkin, "Time series analysis in the time domain and resampling methods for studies of functional magnetic resonance brain imaging," *Human Brain Mapping*, vol. 5, no. 3, pp. 168–193, 1997.

[20] M. J. Brammer, E. T. Bullmore, A. Simmons et al., "Generic brain activation mapping in functional magnetic resonance imaging: a nonparametric approach," *Magnetic Resonance Imaging*, vol. 15, no. 7, pp. 763–770, 1997.

[21] M. Belmonte and D. Yurgelun-Todd, "Permutation testing made practical for functional magnetic resonance image analysis," *IEEE Transactions on Medical Imaging*, vol. 20, no. 3, pp. 243–248, 2001.

[22] E. Bullmore, C. Long, J. Suckling et al., "Colored noise and computational inference in neurophysiological (fMRI) time series analysis: resampling methods in time and wavelet domains," *Human Brain Mapping*, vol. 12, no. 2, pp. 61–78, 2001.

[23] T. E. Nichols and A. P. Holmes, "Nonparametric permutation tests for functional neuroimaging: a primer with examples," *Human Brain Mapping*, vol. 15, no. 1, pp. 1–25, 2002.

[24] T. Nichols and S. Hayasaka, "Controlling the familywise error rate in functional neuroimaging: a comparative review," *Statistical Methods in Medical Research*, vol. 12, no. 5, pp. 419–446, 2003.

[25] S. Hayasaka and T. E. Nichols, "Combining voxel intensity and cluster extent with permutation test framework," *NeuroImage*, vol. 23, no. 1, pp. 54–63, 2004.

[26] M. Breakspear, M. J. Brammer, E. T. Bullmore, P. Das, and L. M. Williams, "Spatiotemporal wavelet resampling for functional neuroimaging data," *Human Brain Mapping*, vol. 23, no. 1, pp. 1–25, 2004.

[27] O. Friman and C. F. Westin, "Resampling fMRI time series," *NeuroImage*, vol. 25, no. 3, pp. 859–867, 2005.

[28] L. Tillikainen, E. Salli, A. Korvenoja, and H. J. Aronen, "A cluster mass permutation test with contextual enhancement for fMRI activation detection," *NeuroImage*, vol. 32, no. 2, pp. 654–664, 2006.

[29] R. Nandy and D. Cordes, "A semi-parametric approach to estimate the family-wise error rate in fMRI using resting-state data," *NeuroImage*, vol. 34, no. 4, pp. 1562–1576, 2007.

[30] A. Eklund, M. Andersson, and H. Knutsson, "fMRI analysis on the GPU-possibilities and challenges," *Computer Methods and Programs in Biomedicine*. In press.

[31] D. Gembris, M. Neeb, M. Gipp, A. Kugel, and R. Männer, "Correlation analysis on GPU systems using NVIDIA's CUDA," *Journal of Real-Time Image Processing*, pp. 1–6, 2010.

[32] A. Eklund, O. Friman, M. Andersson, and H. Knutsson, "A GPU accelerated interactive interface for exploratory functional connectivity analysis of fMRI data," in *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, pp. 1621–1624, 2011.

[33] A. R. Ferreira da Silva, "A bayesian multilevel model for fMRI data analysis," *Computer Methods and Programs in Biomedicine*, vol. 102, pp. 238–252, 2011.

[34] I. Shterev, S.-H. Jung, S. George, and K. Owzar, "permGPU: using graphics processing units in RNA microarray association studies," *BMC Bioinformatics*, vol. 11, p. 329, 2010.

[35] J. L. V. Hemert and J. A. Dickerson, "Monte Carlo randomization tests for large-scale abundance datasets on the GPU," *Computer Methods and Programs in Biomedicine*, vol. 101, no. 1, pp. 80–86, 2011.

[36] K. J. Friston, P. Jezzard, and R. Turner, "Analysis of functional MRI time-series," *Human Brain Mapping*, vol. 1, no. 2, pp. 153–171, 1993.

[37] K. J. Friston, A. P. Holmes, K. J. Worsley, J. P. Poline, C. D. Frith, and R. S. J. Frackowiak, "Statistical parametric maps in functional imaging: a general linear approach," *Human Brain Mapping*, vol. 2, no. 4, pp. 189–210, 1994.

[38] S. J. Kiebel, J. B. Poline, K. J. Friston, A. P. Holmes, and K. J. Worsley, "Robust smoothness estimation in statistical parametric maps using standardized residuals from the general linear model," *NeuroImage*, vol. 10, no. 6, pp. 756–766, 1999.

[39] R. S. Frackowiak, K. Friston, and C. Frith, *Human Brain Function*, Academic Press, New York, NY, USA, 2004.

[40] M. Dwass, "Modified randomization tests for nonparametric hypotheses," *The Annals of Mathematical Statistics*, vol. 28, pp. 181–187, 1957.

[41] A. M. Smith, B. K. Lewis, U. E. Ruttimann et al., "Investigation of low frequency drift in fMRI signal," *NeuroImage*, vol. 9, no. 5, pp. 526–533, 1999.

[42] O. Friman, M. Borga, P. Lundberg, and H. Knutsson, "Detection and detrending in fMRI data analysis," *NeuroImage*, vol. 22, no. 2, pp. 645–655, 2004.

[43] A. R. Laird, B. P. Rogers, and M. E. Meyerand, "Comparison of Fourier and wavelet resampling methods," *Magnetic Resonance in Medicine*, vol. 51, no. 2, pp. 418–422, 2004.

[44] T. Gautama and M. M. Van Hulle, "Optimal spatial regularisation of autocorrelation estimates in fMRI analysis," *NeuroImage*, vol. 23, no. 3, pp. 1203–1216, 2004.

[45] B. Lenoski, L. C. Baxter, L. J. Karam, J. Maisog, and J. Debbins, "On the performance of autocorrelation estimation algorithms for fMRI analysis," *IEEE Journal on Selected Topics in Signal Processing*, vol. 2, no. 6, pp. 828–838, 2008.

[46] H. Knutsson and C.-F. Westin, "Normalized and differential convolution: methods for interpolation and filtering of incomplete and uncertain data," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 515–523, June 1993.

[47] G. M. Ljung and G. E. P. Box, "On a measure of lack of fit in time series models," *Biometrika*, vol. 65, no. 2, pp. 297–303, 1978.

[48] H. Hotelling, "Relation between two sets of variates," *Biometrika*, vol. 28, pp. 322–377, 1936.

[49] T. K. Nguyen, A. Eklund, H. Ohlsson et al., "Concurrent volume visualization of real-time fMRI," in *Proceedings of the 8th IEEE/EG International Symposium on Volume Graphics*, pp. 53–60, Norrköping, Sweden, May 2010.

[50] A. Constantine, "Some non-central distribution problems in multivariate analysis," *Annals of Mathematical Statistics*, vol. 34, pp. 1270–1285, 1963.

[51] S. Das and P. K. Sen, "Restricted canonical correlations," *Linear Algebra and Its Applications*, vol. 210, no. C, pp. 29–47, 1994.

[52] O. Friman, "Subspace models for functional MRI data analysis," in *Proceedings of the 2nd IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pp. 1–4, April 2004.

[53] D. S. Moore, G. P. McCabe, and B. A. Craig, *Introduction to the Practice of Statistics*, W. H. Freeman & Company, 2007.

[54] Nvidia, CUDA Programming Guide, Version 4.0, 2010.

[55] D. Kirk and W. Hwu, *Programming Massively Parallel Processors, A Hands on Approach*, Morgan Kaufmann, 2010.

[56] M. Ragnehed, M. Engström, H. Knutsson, B. Söderfeldt, and P. Lundberg, "Restricted canonical correlation analysis in functional MRI-validation and a novel thresholding technique," *Journal of Magnetic Resonance Imaging*, vol. 29, no. 1, pp. 146–154, 2009.

[57] R. Viviani, P. Beschoner, K. Ehrhard, B. Schmitz, and J. Thöne, "Non-normality and transformations of random fields, with an application to voxel-based morphometry," *NeuroImage*, vol. 35, no. 1, pp. 121–130, 2007.

[58] E. T. Bullmore, J. Suckling, S. Overmeyer, S. Rabe-Hesketh, E. Taylor, and M. J. Brammer, "Global, voxel, and cluster tests, by theory and permutation, for a difference between two groups of structural MR images of the brain," *IEEE Transactions on Medical Imaging*, vol. 18, no. 1, pp. 32–42, 1999.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 403892, 11 pages
doi:10.1155/2011/403892

Research Article

GPU-Accelerated Finite Element Method for Modelling Light Transport in Diffuse Optical Tomography

Martin Schweiger

Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK

Correspondence should be addressed to Martin Schweiger, m.schweiger@cs.ucl.ac.uk

Received 29 March 2011; Revised 1 August 2011; Accepted 4 August 2011

Academic Editor: Yasser M. Kadah

We introduce a GPU-accelerated finite element forward solver for the computation of light transport in scattering media. The forward model is the computationally most expensive component of iterative methods for image reconstruction in diffuse optical tomography, and performance optimisation of the forward solver is therefore crucial for improving the efficiency of the solution of the inverse problem. The GPU forward solver uses a CUDA implementation that evaluates on the graphics hardware the sparse linear system arising in the finite element formulation of the diffusion equation. We present solutions for both time-domain and frequency-domain problems. A comparison with a CPU-based implementation shows significant performance gains of the graphics accelerated solution, with improvements of approximately a factor of 10 for double-precision computations, and factors beyond 20 for single-precision computations. The gains are also shown to be dependent on the mesh complexity, where the largest gains are achieved for high mesh resolutions.

1. Introduction

Diffuse optical tomography (DOT) is a functional imaging modality for medical applications that has the potential to provide three-dimensional images of the scattering and absorption parameter distributions *in vivo*, from which clinically relevant physiological parameters such as tissue and blood oxygenation states and state changes can be derived. Applications include brain activation visualisation [1, 2], brain oxygenation monitoring in infants [3], and breast tumour detection [4].

Data acquisition systems consist of an infrared light delivery system that illuminates the tissue surface at different locations, and detectors that measure the transmitted light at a set of surface positions. Measurements can be performed in continuous wave (CW) mode, in time-resolved mode using ultra-short input pulses and time-resolved detectors, or in frequency-domain mode, using modulated light sources and measuring the phase shift and modulation amplitude at the detector locations.

Due to the high level of scattering in most biological tissues, image reconstruction in DOT is an ill-posed nonlinear problem whose solution generally requires the formulation of a forward model of light propagation in inhomogeneous scattering tissue. Frequently utilised light transport models include stochastic models such as Monte-Carlo simulation [5], or deterministic models such as the radiative transfer equation (RTE) [6] or the diffusion equation (DE) [7]. Numerical solution approaches include finite difference, finite element, finite volume, or boundary element methods. The light transport model considered in this paper the finite element method (FEM) for the solution of the diffusion equation. The reconstruction problem can be stated as a nonlinear optimisation problem, where an objective function, defined as a norm of the difference between measurement data and model data for a given set of optical parameters, is minimised, subject to a regularisation functional. Reconstruction approaches include methods that require the availability of the forward model only, such as Markov-Chain Monte-Carlo methods, its first derivative, such as nonlinear conjugate gradient methods, and its second derivative, such as Newton-type methods.

Iterative solvers require multiple evaluations of the forward model for calculating the objective function and its

gradient. The forward model itself involves the solution of a large linear system with multiple right-hand sides. Problems involving high-dimensional parameter spaces result in time-consuming evaluations of the forward model, which limits the applicability of the reconstruction methods in clinical practice. Significant performance improvements are required to make DOT a viable tool in medical imaging. Recent developments in computing hardware have offered the possibility to make use of parallel computation. Traditionally, solutions have included central processing unit (CPU) based moderately parallel systems with shared memory access (multiprocessor and multicore implementation) and large-scale distributed parallel systems limited by data transfer between nodes (cluster CPU implementation). More recently, the parallel architecture of graphics processing units (GPU) has been utilised for the acceleration of general purpose computations, including GPU methods for the solution of dense [8, 9] or sparse [10–14] linear systems. The latter are encountered in the implementation of the FEM.

In the context of diffuse optical tomography and related fields of optical imaging, GPU-accelerated computations have been successfully employed for implementing Monte-Carlo light transport models [15–17], which compute independent photon trajectories and are well-suited for parallelisation due to the lack of interprocess communication. Acceleration rates of more than 300 are possible. Zhang et al. [18] have applied GPU acceleration to finite element computations in bioluminescence tomography and compared to single and multithreaded CPU performance. They reported significant performance advantages of the GPU version but were limited to low mesh complexity due to memory limits. In optical projection tomography, GPU-based reconstruction methods have been employed by Vinegoni et al. [19]. Watanabe and Itagaki [20] have used a GPU implementation for real-time visualisation in Fourier-domain optical coherence tomography.

In this paper, we are investigating the potential of a GPU implementation for the forward model in DOT. We present a Compute Unified Device Architecture (CUDA) version of the finite element forward solver presented previously [21, 22], using the CUSP library [23] for sparse linear system computation on the graphics processor. CUDA is the computing architecture for NVidia graphics processors and can be addressed via an application-programming interface (API). Current GPU hardware is performance optimised for single-precision arithmetic. We investigate the effect of single-precision computation on the accuracy of the forward model for different combinations of optical parameters. We compare the performance of the GPU forward solver with an equivalent CPU implementation. We show that significant performance improvements can be achieved. The evaluation of the forward model is the most time-consuming element of iterative inverse solvers, and any efficiency gains in the forward solver therefore directly translate into reduced overall runtimes for image reconstruction applications and are an important step towards making DOT a viable imaging application in clinical practice.

2. Methodology

2.1. Finite Element Solver. We consider the diffusion approximation to the radiative transfer equation [24, 25] in either steady-state, time, or frequency domain as the forward model for light transport in tissue. For steady-state problems, the stationary real-valued photon density inside the medium arising from a continuous-wave source is computed while for frequency-domain problems, the source is amplitude modulated, giving rise to a complex-valued solution of a photon density wave distribution. In time-domain problems, the source is considered a delta-pulse in time, and the measurement consists of the temporal dispersion of the transmitted signal. Given a compact domain Ω bounded by $\partial\Omega$, the diffusion equation [26] in time and frequency domain is given by

$$\left.\begin{aligned}\left[-\nabla\cdot\kappa(\mathbf{r})\nabla+\mu_a(\mathbf{r})+\frac{1}{c}\frac{\partial}{\partial t}\right]\phi(\mathbf{r},t)&=0\\ \left[-\nabla\cdot\kappa(\mathbf{r})\nabla+\mu_a(\mathbf{r})+\frac{i\omega}{c}\right]\hat{\phi}(\mathbf{r},\omega)&=0\end{aligned}\right\}\mathbf{r}\in\Omega, \quad (1)$$

respectively, where ω is the angular source modulation frequency, $\kappa(\mathbf{r})$ and $\mu_a(\mathbf{r})$ are the spatially varying diffusion and absorption coefficients, respectively, where $\kappa = [3(\mu_a + \mu_s)]^{-1}$ with scattering coefficient μ_s. c is the speed of light in the medium, and ϕ, and $\hat{\phi}$ are the real and complex-valued photon density fields. For simplicity in the following, we use ϕ to denote either the real or complex-valued properties as appropriate.

A Robin-type boundary condition [27] applies at $\partial\Omega$,

$$\phi(\xi)+2\zeta(n)\kappa(\xi)\frac{\partial\phi}{\partial\nu}=q(\boldsymbol{\xi}),\quad \xi\in\partial\Omega, \quad (2)$$

where q is a real or complex-valued source distribution as appropriate, $\zeta(n)$ is a boundary reflectance term incorporating the refractive index n at the tissue-air interface, and ν is the surface normal at surface point ξ. The boundary operator defining the exitance Γ through $\partial\Omega$ is given by the Dirichlet-to-Neumann map

$$\Gamma(\xi)=-c\kappa(\xi)\frac{\partial\phi}{\partial\nu}=\frac{c}{2\zeta}\phi(\xi). \quad (3)$$

The set of measurements y_{ij} from a source distribution q_i is obtained by integrating Γ over the measurement profiles $m_j(\xi)$ on the surface

$$y_{ij}=\int_{\partial\Omega}\Gamma_i(\xi)m_j(\xi)d\xi. \quad (4)$$

For the time-domain problem, y_{ij} are the temporal dispersion profiles of the received signal intensities while, for the frequency-domain problem, y_{ij} are given by the complex exitance values, usually expressed by logarithmic amplitude $\ln A$ and phase shift φ [28],

$$\ln A_{ij}=\mathrm{Re}\left(\ln y_{ij}\right),\qquad \varphi_{ij}=\mathrm{Im}\left(\ln y_{ij}\right). \quad (5)$$

Given the set of forward data $\mathbf{y}=\{y_{ij}\}$ of all measurements from all source distributions, (1) to (4) define the forward

model $f[\kappa, \mu_a] = \mathbf{y}$ which maps a parameter distribution κ, μ_a to measurements for a given domain geometry, modulation frequency, source distributions, and measurement profiles.

The forward model is solved numerically by using a finite element approach. A division of domain Ω into tetrahedral elements defined by N vertex nodes provides a piecewise polynomial basis for the parameters κ, μ_a, and photon density ϕ. The approximate field $\phi^h(\mathbf{r})$ at any point $\mathbf{r} \in \Omega$ is given by interpolation of the nodal coefficients ϕ_i using piecewise polynomial shape functions $u_i(\mathbf{r})$

$$\phi^h(\mathbf{r}) = \sum_{i=1}^{N} u_i(\mathbf{r})\phi_i. \tag{6}$$

Piecewise polynomial approximations κ^h, μ_a^h to the continuous parameters, defined by the nodal coefficients κ_i, $\mu_{a,i}$ are constructed in the same way. Applying a Galerkin approach transforms the continuous problem of (1) into an N-dimensional discrete problem of finding the nodal field values $\Phi = \{\phi_i\}$ at all nodes i, given the set of nodal parameters $\mathbf{x} = \{\kappa_i, \mu_{a,i}\}$. For the frequency-domain problem, the resulting linear system is given by

$$\mathsf{S}(\mathbf{x}, \omega)\Phi(\omega) = Q(\omega), \tag{7}$$

where

$$\mathsf{S}(\mathbf{x}, \omega) = \mathsf{K}(\{\kappa_i\}) + \mathsf{C}(\{\mu_{a,i}\}) + \gamma\mathsf{A} + i\omega\mathsf{B}, \tag{8}$$

$\gamma = c/2\zeta$, $\mathsf{K}, \mathsf{C}, \mathsf{A}, \mathsf{B} \in \mathbb{R}^{N\times N}$ are symmetric sparse matrices given by [7]

$$\begin{aligned}
K_{ij} &= \sum_{k=1}^{N} \kappa_k \int_\Omega u_k(\mathbf{r})\nabla u_i(\mathbf{r}) \cdot \nabla u_j(\mathbf{r}) d\mathbf{r},\\
C_{ij} &= \sum_{k=1}^{N} \mu_{a,k} \int_\Omega u_k(\mathbf{r}) u_i(\mathbf{r}) u_j(\mathbf{r}) d\mathbf{r},\\
A_{ij} &= \int_{\partial\Omega} u_i(\boldsymbol{\xi}) u_j(\boldsymbol{\xi}) d\boldsymbol{\xi},\\
B_{ij} &= \frac{1}{c} \int_\Omega u_i(\mathbf{r}) u_j(\mathbf{r}) d\mathbf{r}.
\end{aligned} \tag{9}$$

And right-hand side Q is given by

$$Q_i = \sum_{k=1}^{N} q_i \int_{\partial\Omega} u_i(\boldsymbol{\xi}) d\boldsymbol{\xi} \tag{10}$$

with q_i the nodal coefficients of the basis expansion of $q(\boldsymbol{\xi})$. For the solution of the time-domain problem, the time derivative in (1) at time t is approximated by a finite difference

$$\frac{\partial\phi\left(\vec{r}, t\right)}{\partial t} \approx \frac{1}{\Delta t}\left[\phi\left(\vec{r}, t + \Delta t\right) - \phi\left(\vec{r}, t\right)\right]. \tag{11}$$

The temporal profile of ϕ is approximated at a set of discrete steps $\{t_n\}$ and evaluated by a finite difference approach, given by the iteration

$$\begin{aligned}
&\left[\theta\tilde{\mathsf{S}} + \frac{1}{\Delta t_0}\mathsf{B}\right]\Phi(t_0) = \frac{1}{\Delta t_0}\mathbf{Q}_0,\\
&\left[\theta\tilde{\mathsf{S}} + \frac{1}{\Delta t_n}\mathsf{B}\right]\Phi(t_n) = -\left[(1-\theta)\tilde{\mathsf{S}} - \frac{1}{\Delta t_n}\mathsf{B}\right]\Phi(t_{n-1}), \quad n \geq 1,
\end{aligned} \tag{12}$$

where $\tilde{\mathsf{S}} = \mathsf{K} + \mathsf{C} + \gamma\mathsf{A}$, time steps $t_n = t_{n-1} + \Delta t_{n-1}$, $n \geq 1$, and $0 \leq \theta \leq 1$ is a control parameter that can be used to select implicit ($\theta = 1$), explicit ($\theta = 0$), or intermediate schemes. The step lengths Δt_n are governed by stability considerations of the finite difference scheme. For the unconditionally stable implicit scheme, the step length can be adjusted to the curvature of the temporal profile, allowing increased step length at the exponentially decaying tail of $\phi(t)$.

The solution of the FEM problem thus consists of (i) construction of the system matrices (9), (ii) solution of the complex-valued linear problem (7) or real-valued sequence of linear problems (12), and (iii) mapping to measurements (3) and (4). The main computational cost is the solution of the linear system, in particular in the time-domain problem, while the cost of matrix assembly time is typically only 1–10% of the time of a single linear solution. The linear system can be solved either with a direct method, such as Cholesky decomposition for the real-valued time-domain problem or LU decomposition for the complex-valued frequency domain problem, or with iterative methods, such as conjugate gradients for the real-valued problem and biconjugate gradients for the complex-valued problem. Direct methods become impractical for large-scale problems, due to memory storage requirements for the decomposition and increased computation time. For 3-D problems with high node density, iterative solvers are generally employed.

2.2. GPU Implementation. The bottleneck of the reconstruction problem is the solution of the linear systems in (7) or (12). Accelerating the linear solver is therefore an effective method for improving the inverse solver performance. We have embedded a graphics processor-accelerated version of the FEM forward solver into the existing TOAST software package [29] for light transport and reconstruction presented previously [7]. The GPU-accelerated code uses the CUDA programming interface for NVidia graphics processor hardware. The implementation utilises the CUSP library which offers a templated framework for sparse linear algebra and provides conjugate gradient (CG) and biconjugate gradient-stabilised (BiCGSTAB) iterative solvers for sparse linear systems. The library supports both single and double precision computation if supported by hardware.

We use the compressed sparse row (CSR) format for matrix storage. There are alternative storage formats such as the coordinate, ELLPACK, or hybrid formats [11] which can provide better parallel performance depending on the matrix fill structure, usually at the cost of less compact storage.

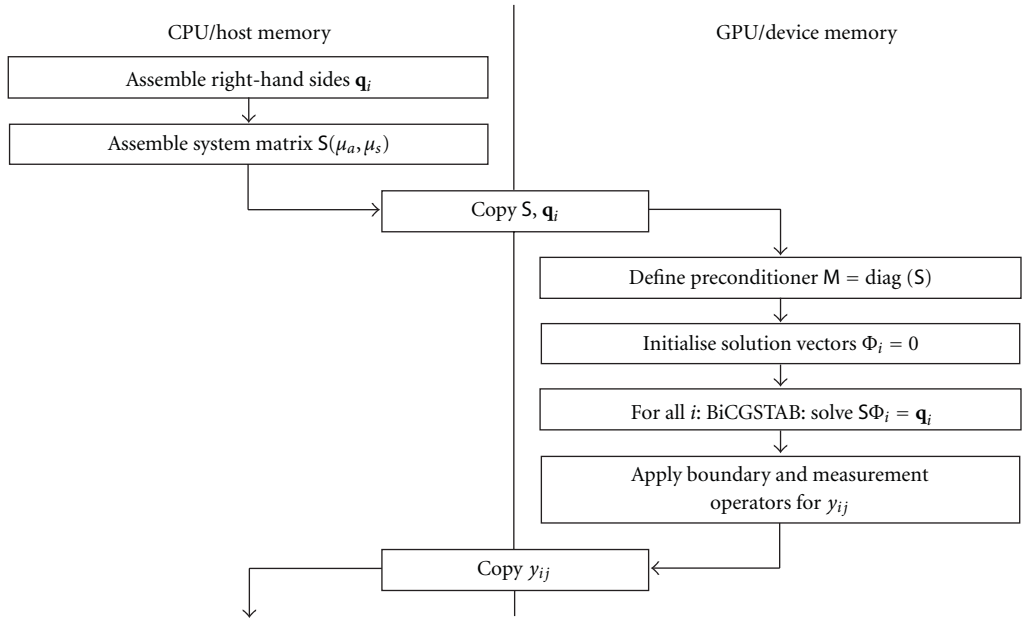


Figure 1: Data flow between host and graphics device for solution of linear problem (7).

However, the CSR format constitutes a good compromise between performance and versatility and is well suited for the matrix fill distribution arising from unstructured FEM meshes.

For the solution of the complex-valued linear problem (7), we expand the complex $N \times N$ system into a $2N \times 2N$ real system of the form

$$\begin{bmatrix} \mathsf{S}_{\text{re}} & -\mathsf{S}_{\text{im}} \\ \mathsf{S}_{\text{im}} & \mathsf{S}_{\text{re}} \end{bmatrix} \begin{bmatrix} \Phi_{\text{re}} \\ \Phi_{\text{im}} \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_{\text{re}} \\ \mathbf{Q}_{\text{im}} \end{bmatrix}. \tag{13}$$

The CUSP CG and BiCGSTAB solvers had to be modified to account for early termination of the iteration loop due to singularities in the intermediate results. Early termination conditions occasionally do occur in practice in the problems considered in this paper, in particular due to single-precision round-off errors.

The data flow between host and graphics device memory for a single solver step is shown in Figure 1. The system matrix S is assembled in host memory for a given set of parameters, together with the source vectors $\mathbf{q}_i$, and copied to GPU device memory. The GPU solver is then invoked for all right-hand-sides, after which the projected solutions y_{ij} are copied back to host memory.

For the finite-difference solution of the time-domain problem, the entire iteration (12) can be evaluated on the GPU with minimal communication between host and graphics system, consisting of initial copying the system matrices $\tilde{\mathsf{S}}$ and B to the GPU, and returning the computed temporal profiles $y_{ij}(t)$ back to the host. The data flow diagram for the time-domain problem is shown in Figure 2.

2.3. Single-Precision Arithmetic. GPU hardware is traditionally optimised for single-precision floating point operations. Although GPU hardware with double-precision capability is emerging, typically only a fraction of the chip infrastructure is dedicated to double-precision operations, thus incurring a significant performance penalty. For optimising, it is therefore advantageous to use single-precision arithmetic where adequate. We have implemented the FEM solver in both single and double precision for GPU as well as CPU platforms.

When the system matrix is represented in single precision, care has to be taken during assembly. The system matrix is assembled from individual element contributions (9). The global vertex contributions in the system matrix are the sum of the local element vertex values for all elements adjacent to the vertex. During the summation, loss of precision can occur if the magnitude difference between the summands is large compared to the mantissa precision of the floating point representation. For single precision arithmetic, this can be a problem in particular where vertices have a large number of adjacent elements, notably in 3-D meshes with tetrahedral elements. Loss of precision during matrix assembly can be reduced if the contributions are sorted from smallest to highest magnitude. However, this incurs a book-keeping overhead that can impact on performance. Instead we have opted to assemble the system matrix in double precision and map the values to single precision after assembly. The assembly step is performed on the host side, with negligible performance impact because assembly time is generally small compared to solve time.

To compare the results of matrix assembly in single and double precision, we have performed an FEM forward solution from single-precision system matrices that were assembled in both single and double precision. The domain was a homogeneous cylinder of radius 25 mm and height 50 mm, with optical parameters $\mu_a = 0.01\,\text{mm}^{-1}$ and $\kappa = 0.3\,\text{mm}$.

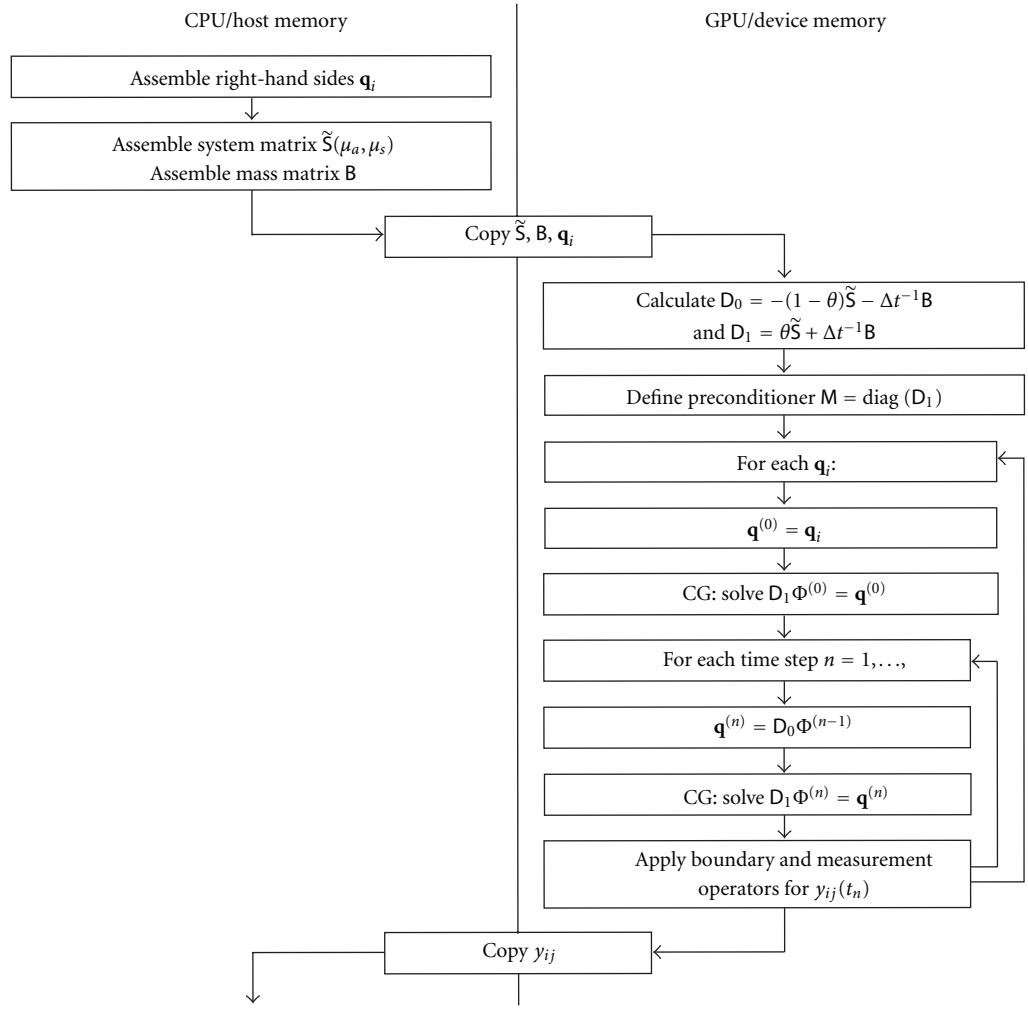


Figure 2: Data flow between host and graphics device for solution of linear problem (12).

A point source modulated at frequency $\omega = 2\pi \cdot 100\,\mathrm{MHz}$ was placed on the cylinder mantle. The mesh consisted of 83142 and 444278 tetrahedral elements.

Figure 3 shows the differences between single and double precision forward solution in log amplitude (Figure 3(a)) and phase (Figure 3(b)) of the complex photon density field along a line from the source position across the volume of the cylinder. The solid lines represent the single-precision error where the system matrix has been assembled in double precision before being mapped to single precision, while the dashed line is the error arising from a system matrix assembled in single precision. It can be seen that system matrix assembly in double precision can significantly reduce the solution errors, in particular at large distances from the source.

The influence of optical parameters on the single-precision error of the forward data is shown in Figure 4. The forward solutions were calculated for three different combinations of absorption and scattering coefficient (i) $\mu_a = 0.01\,\mathrm{mm}^{-1}$, $\mu_s = 1\,\mathrm{mm}^{-1}$, (ii) $\mu_a = 0.1\,\mathrm{mm}^{-1}$, $\mu_s = 1\,\mathrm{mm}^{-1}$, and (iii) $\mu_a = 0.1\,\mathrm{mm}^{-1}$, $\mu_s = 1.5\,\mathrm{mm}^{-1}$. It can be seen that the discrepancies become more severe at higher values of the optical parameters. The results are particularly sensitive to an increase of the scattering parameter. Due to attenuation, the photon density fields inside the object decay rapidly, leading to large dynamic range in the data. Increased absorption and scattering parameters aggravate this effect, which impairs the accuracy of the single-precision solution, in particular in regions far away from the source. It should be noted, however, that, for moderate optical parameters in a typical range for optical tomography, the single precision solution is accurate, with maximum relative errors of 10^{-6} to 10^{-4} in log amplitude and phase, respectively.

3. Results

The graphics accelerated forward solver problems were executed on an NVidia GTX 285 GPU. The technical specifications of the device are listed in Table 1. The device supports double as well as single precision arithmetic, so results for both were collected. For performance comparison, the same model calculations were also performed with a CPU-based serial implementation on an Intel Xeon processor clocked

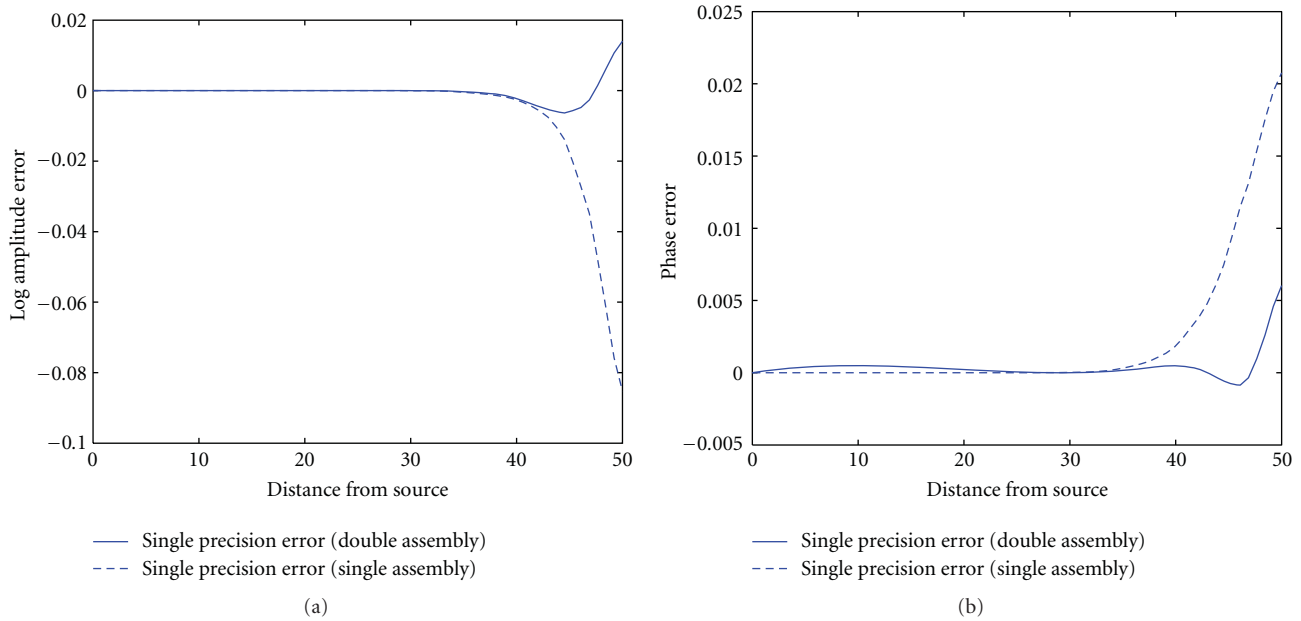


FIGURE 3: Effect of single-precision arithmetic on forward solutions. Shown are the differences between single and double precision solutions for logarithmic amplitude (a) and phase (b) of the complex field computed in a cylindrical domain along a line from the source across the cylinder. The solid line shows the solution error for a system matrix assembled in double precision and solved in single precision while the dashed line represents the solution for a system matrix assembled and solved in single precision.

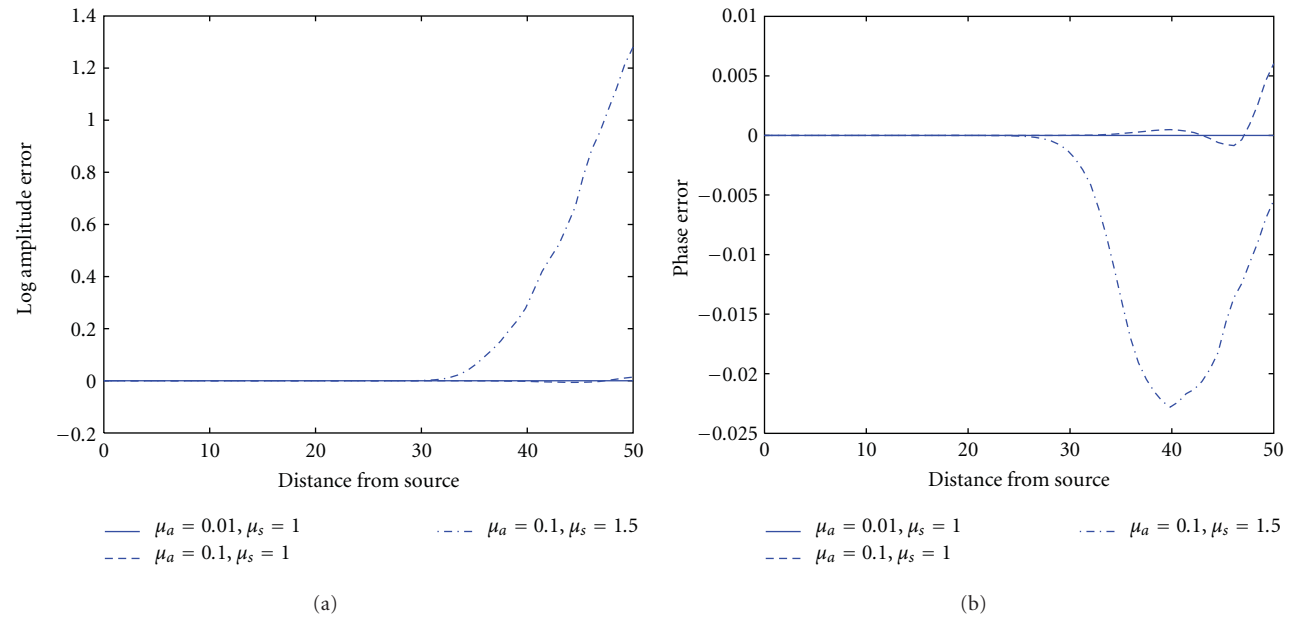


FIGURE 4: Single-precision arithmetic error as a function of optical coefficients. Shown are the differences between single and double precision results in log amplitude (a) and phase (b) along a line from the source across the cylinder, for three different combinations of absorption and scattering parameters.

at 2.0 GHz with 4 MB cache and 12 GB main memory. The FEM model consisted of a homogeneous cylindrical mesh with radius 25 mm and height 50 mm at various element resolutions. 80 sources and 80 detectors were arranged on the surface. One of the meshes, together with the resulting system matrix sparsity structure, is shown in Figure 5.

3.1. Frequency Domain Solver. For run-time performance comparison between GPU and CPU implementations under a variety of conditions, we evaluated the frequency-domain FEM forward model using different mesh complexities. The forward solutions were computed for both a complex-valued problem using a modulation frequency of $\omega = 2\pi \cdot 100$ MHz

Table 1: Computational capabilities of GPU platform.

Platform	GeForce GTX 285
Global device memory	1 GB
Processor core clock	1.476 GHz
Memory clock	1.242 GHz
CUDA cores	240
Multiprocessors	30

and a real-valued steady-state problem of $\omega = 0$. For the complex-valued problem, the BiCGSTAB linear solver was used to compute the linear system in (7) while, for the real-valued problem a CG solver was used. We tested the performance of the GPU solution as a function of the CG and BiCGSTAB convergence tolerances, either without preconditioner or with a diagonal preconditioner. The results are shown in terms of the GPU performance factor, given by the ratio of the CPU and GPU run times. Figure 6 shows the performance factors for single precision (Figure 6(a)) and double precision (Figure 6(b)) calculations. It can be seen that the GPU achieves a performance factor between 8 and 19 for single precision calculations, depending on the problem type, where the real-valued BiCGSTAB solution without preconditioner shows the highest improvement at 14–19, while the complex BiCGSTAB solution without preconditioner exhibits the smallest improvement at 8–11.5. Generally, the performance factor drops for lower tolerance limits. The performance factors for double-precision solutions are significantly lower, in a range between 3.7 and 4.7. This is due to the fact that while GPU performance drops significantly for double-precision calculations, the CPU solver performance is generally not affected, and indeed the CPU performance is slightly higher at double precision because it avoids casting floating point buffers between single and double precision. The drop in performance factor for lower tolerance limits is not present in the double-precision results.

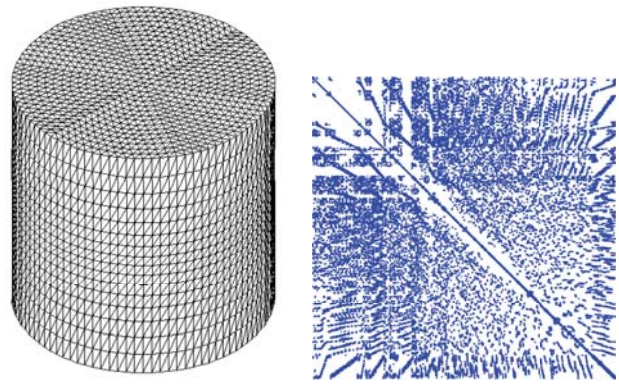
The next test compares the CPU and GPU performance as a function of the mesh node density and the resulting size of the linear system. The performance factors for the forward solvers applied to cylindrical meshes of different mesh resolutions as a function of node count are shown in Figure 7. At each mesh resolution, we solved both a real-valued steady-state problem with the preconditioned CG solver, and a complex-valued frequency-domain problem with the preconditioned BiCGSTAB solver, at single-precision (Figure 7(a)) and double-precision (Figure 7(b)) resolution. All solver results are for calculating the real or complex photon density fields for 80 sources, for a solver tolerance fixed at 10^{-10}. It can be seen that in all cases GPU performance improves with increasing size of the linear system. For the single-precision solver, the performance factors range between 1 and 26 for mesh node counts between 9000 and $3.3 \cdot 10^5$, respectively, for the steady-state problem, and between 2 and 30 for mesh node counts between 9000 and $2.5 \cdot 10^5$, respectively, for the frequency domain problem. Note that for the frequency domain problem, the performance factors could not be computed for the two largest meshes due to excessive computation time of the CPU solution. The absolute linear solver times for selected cases are shown in Table 2. It can be seen that for the largest mesh resolutions, forward solver times on the CPU can take in excess of an hour. This can be prohibitive for clinical applications in iterative reconstruction, where each step of the reconstruction may require multiple evaluations of the forward problem to calculate the objective function and its gradient at the current estimate or perform a line search along the current search direction. By comparison, the GPU times for these problems typically require 2 to 10 minutes, which is feasible for reconstruction problems.

To provide a comparison with a CPU-based parallel solver, we also show the performance factors of a shared-memory thread-based version of the FEM forward solver using up to 8 threads, compared to the single-thread serial implementation. The thread implementation uses a coarse-grain parallelisation strategy, dividing the solution of the linear problems for different right-hand sides over the available worker threads. This method provided better processor utilisation and less communication overhead for the problem considered here than a fine-grain strategy of parallelising the iterative solver itself. Because the CPU implementation showed no significant performance difference between the single and double precision solution, we present here only the double-precision results. Figure 8 shows the performance factors for 2, 4, and 8 threads for the real-valued problem using a CG solver, and for the complex-valued problem using a BiCGSTAB solver. The CG solver reaches factors between 1.5 (2 threads) and 2.8 (8 threads) while the BiCGSTAB solver reaches factors between 1.7 (2 threads) and 4 (8 threads). The dependency on mesh complexity is not as marked as for the GPU solver.

3.2. Time-Domain Solver. We computed the finite difference implementation of the time-domain problem (12) over 100 time steps of 50 picoseconds for cylinder meshes of different complexity. For these simulations, a Crank-Nicholson scheme ($\theta = 0.5$) was used. Signal intensity time profiles were calculated at 80 detector position for each of 80 source locations. The performance results are shown in Figure 9. It can be seen that the performance improvements of the GPU implementation is again strongly dependent on mesh resolution, ranging from a factor of 3 to 13 for the double-precision arithmetic calculation, and from 6 to 17 for the single-precision calculation. At the highest mesh resolution, the total forward solver run time is approximately 8 hours for the CPU implementation for both single and double precision while the GPU run time is approximately 29 and 36 minutes for the single and double precision solutions, respectively.

4. Conclusions

We have developed a GPU implementation of a finite element forward model for diffuse light transport that can be used as a component in an iterative nonlinear reconstruction method in diffuse optical tomography. The efficiency of the forward solver has a significant impact on reconstruction



(a) (b)

Figure 5: Cylinder geometry for the forward and inverse solver problems, showing a mesh with 83142 nodes and 444278 tetrahedral elements (b). The fill structure of the resulting FEM system matrix is shown on the right. The number of nonzeros is 1150264, resulting in a fill fraction of $1.664 \cdot 10^{-4}$.

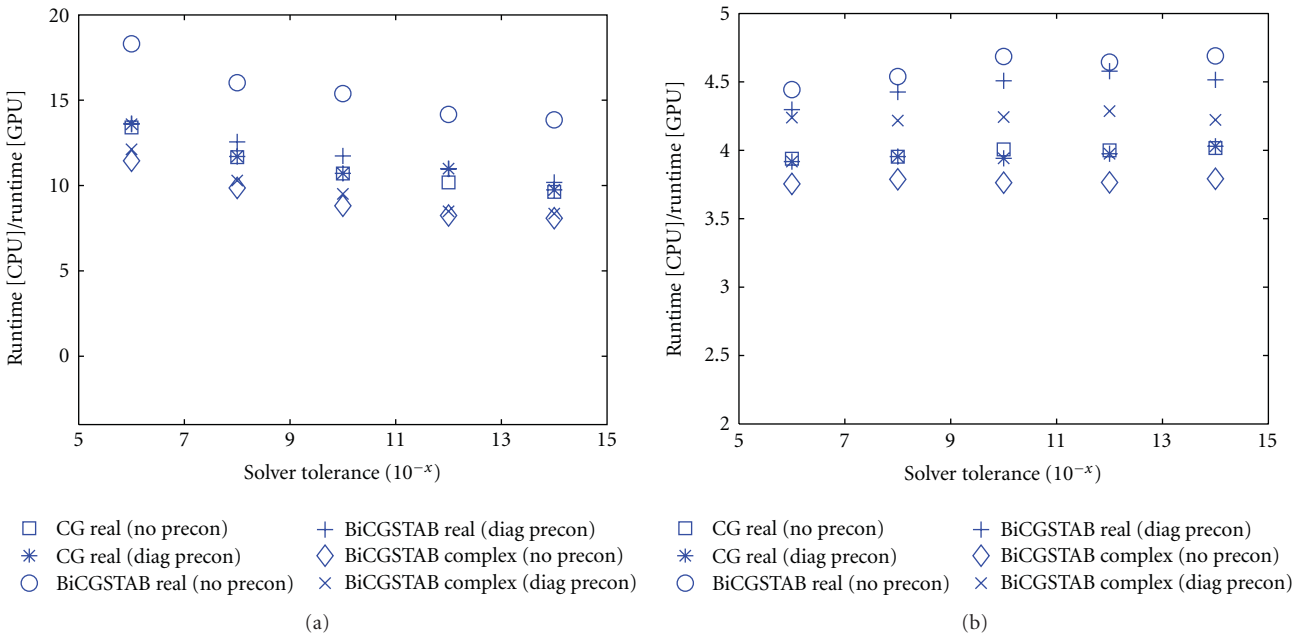


Figure 6: GPU performance factor as a function of linear solver tolerance for real and complex problems, using CG and BiCGSTAB solvers, without preconditioner and with diagonal preconditioner. (a): single-precision performance, (b): double-precision performance.

Table 2: GPU run-time comparisons for FEM forward solver computations of 80 source distributions in cylindrical meshes of different node densities. Real-valued problems were solved with a conjugate gradient solver, complex problems with a biconjugate gradient stabilised solver. Values in parentheses are CPU solution times.

Node number	Runtime [s]			
	Real single	Complex single	Real double	Complex double
8987	11.07	13.43	11.1	14.4
	(8.49)	(26.15)	(3.74)	(10.46)
82517	23.24	60.03	26.85	75.42
	(193.94)	(678.65)	(96.21)	(271.9)
245917	82.56	433.89	117.41	523.76
	(1789.7)	(12996.8)	(837.39)	(2351.25)
327617	127.19	1060.42	189.06	909.45
	(3258.46)	(−)	(1509.22)	(4699.71)

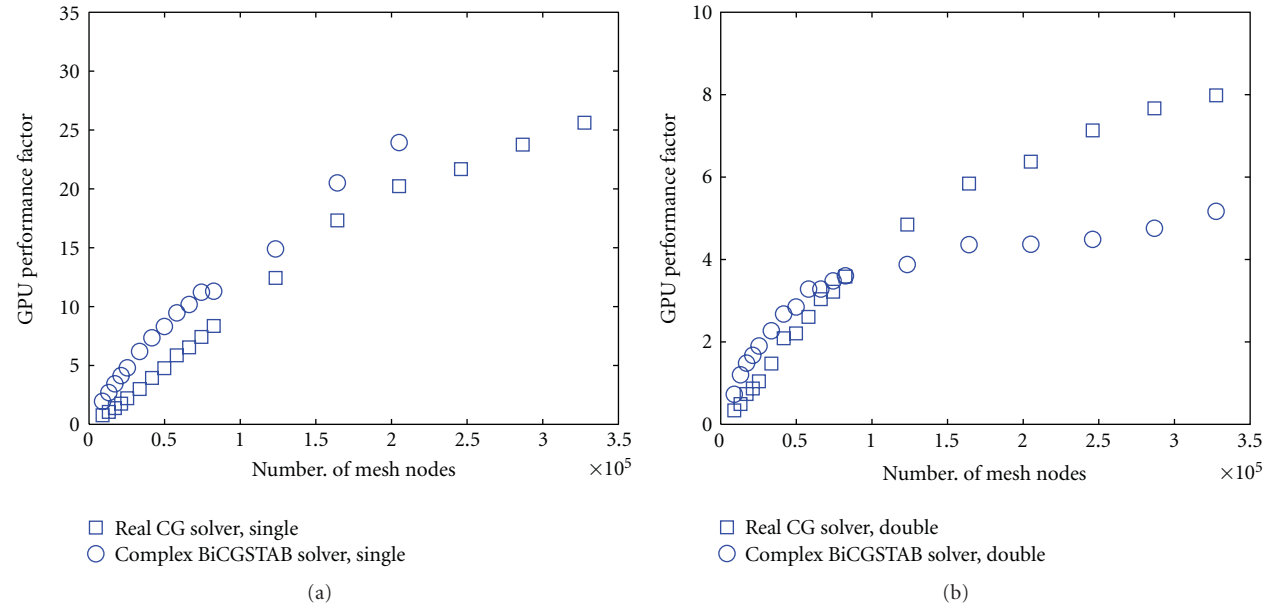


Figure 7: GPU performance factor as a function of mesh node count for a real-valued problem solved with preconditioned CG solver, and a complex-valued problem solved with preconditioned BiCGSTAB solver. (a): single-precision performance, (b): double-precision performance.

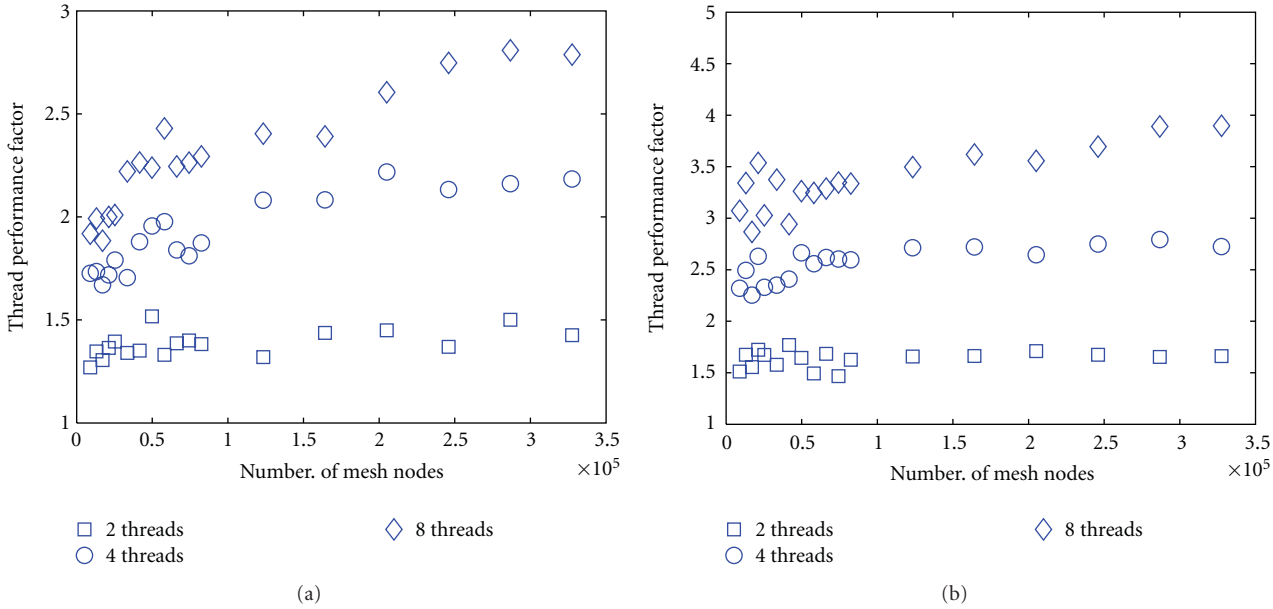


Figure 8: Performance factors for CPU-threaded versus CPU-serial forward solver computations as a function of node densities. (a): CG solver for real-valued problem, (b): BiCGSTAB solver for complex-valued problem.

performance, and the reduction of reconstruction times is essential in making optical tomography a viable imaging modality in clinical diagnosis.

The model presented here supports real and complex-valued problems and can be applied to steady-state, time, or frequency-domain imaging systems. The linear system arising from the FEM discretisation is solved either with a conjugate gradient or biconjugate gradient stabilised iterative solver on the GPU device. We have shown that the GPU solver can achieve significant performance improvements over a serial CPU implementation in the range of factors between 5 and 30, depending on mesh complexity, tolerance limit, and solver type. The GPU-based forward solver provides higher performance gains than a thread-based parallel

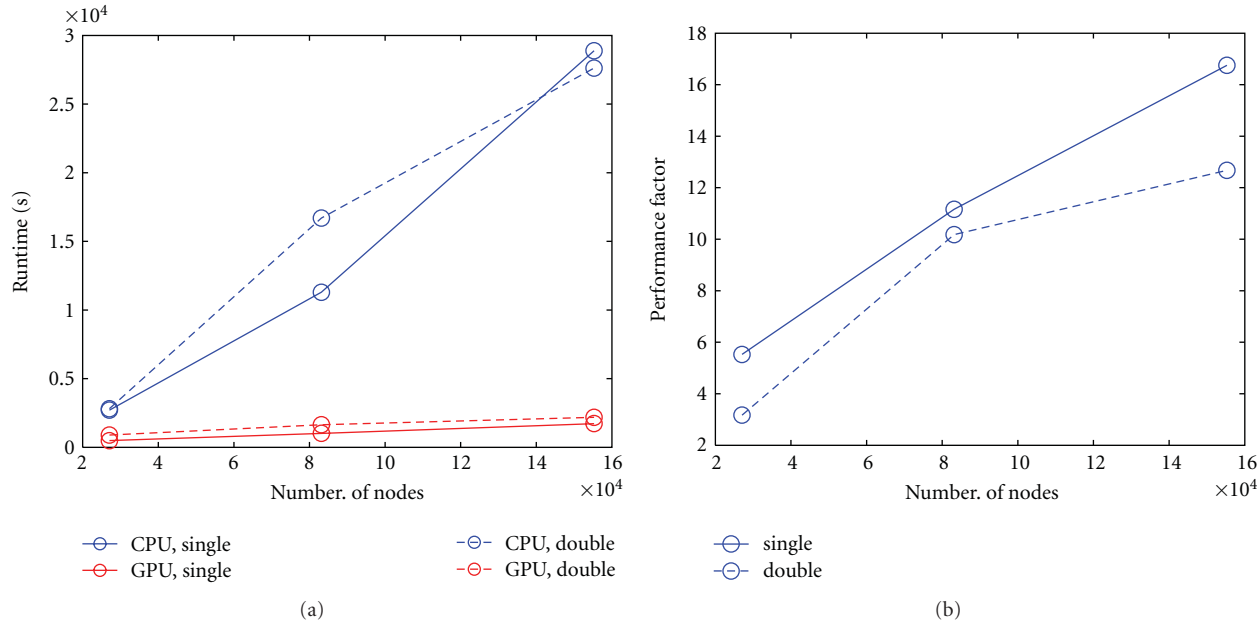


Figure 9: Run-time comparison for CPU and GPU forward solution of time-dependent FEM problem over 100 time steps. (a): Run-times for single and double precision arithmetic as a function of mesh complexity; (b): performance factors.

CPU implementation that was used for comparison. Future developments in GPU hardware are expected to increase the performance gain even further.

We have shown that for the forward problem a single precision linear solver can be applied for typical ranges of optical parameters in clinical applications of optical parameters. Single-precision arithmetic yields higher performance in particular for GPU-computing platforms. However, at very high absorption and scattering parameter values, the linear system may become increasingly ill-conditioned and no longer converge with single-precision arithmetic. In these cases, double-precision computation is required.

Acknowledgments

This work was supported by EPSRC Grant EP/E034950/1 and the EC Seventh Framework Programme (FP7/2007–2013) under Grant agreement no. 201076.

References

[1] D. A. Boas, D. H. Brooks, E. L. Miller et al., "Imaging the body with diffuse optical tomography," *IEEE Signal Processing Magazine*, vol. 18, no. 6, pp. 57–75, 2001.

[2] B. W. Pogue, K. D. Paulsen, C. Abele, and H. Kaufman, "Calibration of near-infrared frequency-domain tissue spectroscopy for absolute absorption coefficient quantitation in neonatal head-simulating phantoms," *Journal of Biomedical Optics*, vol. 5, no. 2, pp. 185–193, 2000.

[3] M. Cope and D. T. Delpy, "System for long-term measurement of cerebral blood and tissue oxygenation on newborn infants by near infra-red transillumination," *Medical and Biological Engineering and Computing*, vol. 26, no. 3, pp. 289–294, 1988.

[4] D. J. Hawrysz and E. M. Sevick-Muraca, "Developments toward diagnostic breast cancer imaging using neer-infrared optical measurements and fluorescent contrast agents," *Neoplasia*, vol. 2, no. 5, pp. 388–417, 2000.

[5] D. A. Boas, J. P. Culver, J. J. Stott, and A. K. Dunn, "Three dimensional Monte Carlo code for photon migration through complex heterogeneous media including the adult human head," *Optics Express*, vol. 10, no. 3, pp. 159–170, 2002.

[6] G. S. Abdoulaev and A. H. Hielscher, "Three-dimensional optical tomography with the equation of radiative transfer," *Journal of Electronic Imaging*, vol. 12, no. 4, pp. 594–601, 2003.

[7] M. Schweiger, S. R. Arridge, and I. Nissilä, "Gauss-Newton method for image reconstruction in diffuse optical tomography," *Physics in Medicine and Biology*, vol. 50, no. 10, pp. 2365–2386, 2005.

[8] A. Moravánsky and N. Ag, "Dense matrix algebra on the GPU," in *Direct3D ShaderX2*, W. F. Engel, Ed., p. 2, Wordware Publishing, Plano, Tex,USA, 2003.

[9] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware," in *Proceedings of the ACM/IEEE SC 2005 Conference*, p. 3, IEEE Computer Society, Washington, DC, USA, December 2005.

[10] J. D. Owens, D. Luebke, N. Govindaraju et al., "A survey of general-purpose computation on graphics hardware," in *Proceedings of the Computer Graphics Forum*, vol. 34, pp. 80–113, Blackwell Publishing, March 2007.

[11] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Tech. Rep. NVR-2008-004, NVIDIA Corporation, 2008.

[12] L. Buatois, G. Caumon, and B. Levy, "Concurrent number cruncher: an efficient sparse linear solver on the GPU," in *Proceedings of the 3rd International Conference High Performance Computing and Communications, (HPCC'07)*, vol. 4782

of *Lecture Notes in Computer Science*, pp. 358–371, Springer, Houston, Tex, USA, September 2007.

[13] J. Kráger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Transactions on Graphics*, vol. 22, pp. 908–916.

[14] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Transactions on Graphics*, vol. 22, pp. 917–924.

[15] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration," *Journal of Biomedical Optics*, vol. 13, no. 6, Article ID 060504, 2008.

[16] Q. Fang and D. A. Boas, "Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units," *Optics Express*, vol. 17, no. 22, pp. 20178–20190, 2009.

[17] N. Ren, J. Liang, X. Qu, J. Li, B. Lu, and J. Tian, "GPU-based Monte Carlo simulation for light propagation in complex heterogeneous tissues," *Optics Express*, vol. 18, no. 7, pp. 6811–6823, 2010.

[18] B. Zhang, X. Yang, F. Yang et al., "The CUBLAS and CULA based GPU acceleration of adaptive finite element framework for bioluminescence tomography," *Optics Express*, vol. 18, no. 19, pp. 20201–20213, 2010.

[19] C. Vinegoni, L. Fexon, P. F. Feruglio et al., "High throughput transmission optical projection tomography using low cost graphics processing unit," *Optics Express*, vol. 17, no. 25, pp. 22320–22332, 2009.

[20] Y. Watanabe and T. Itagaki, "Real-time display on Fourier domain optical coherence tomography system using a graphics processing unit," *Journal of Biomedical Optics*, vol. 14, no. 6, Article ID 060506, 2009.

[21] S. R. Arridge, M. Schweiger, M. Hiraoka, and D. T. Delpy, "A finite element approach for modeling photon transport in tissue," *Medical Physics*, vol. 20, no. 2, pp. 299–309, 1993.

[22] M. Schweiger and S. R. Arridge, "The finite-element method for the propagation of light in scattering media: frequency domain case," *Medical Physics*, vol. 24, no. 6, pp. 895–902, 1997.

[23] Cusp library, http://code.google.com/p/cusp-library/.

[24] S. R. Arridge, "Optical tomography in medical imaging," *Inverse Problems*, vol. 15, no. 2, pp. R41–R93, 1999.

[25] M. S. Patterson, B. Chance, and B. C. Wilson, "Time resolved reflectance and transmittance for the non-invasive measurement of tissue optical properties," *Applied Optics*, vol. 28, no. 12, pp. 2331–2336, 1989.

[26] A. Ishimaru, *Wave Propagation and Scattering in Random Media*, vol. 1, Academic Press, New York, NY, USA, 1978.

[27] R. C. Haskell, L. O. Svaasand, T. T. Tsay, T. C. Feng, M. S. McAdams, and B. J. Tromberg, "Boundary conditions for the diffusion equation in radiative transfer," *Journal of the Optical Society of America A*, vol. 11, no. 10, pp. 2727–2741, 1994.

[28] I. Nissilä, K. Kotilahti, K. Fallström, and T. Katila, "Instrumentation for the accurate measurement of phase and amplitude in optical tomography," *Review of Scientific Instruments*, vol. 73, no. 9, pp. 3306–3312, 2002.

[29] M. Schweiger and S. R. Arridge, "Toast reconstruction package," http://toastplusplus.org.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 680765, 16 pages
doi:10.1155/2011/680765

Research Article

Numerical Solution of Diffusion Models in Biomedical Imaging on Multicore Processors

Luisa D'Amore,[1] Daniela Casaburi,[2] Livia Marcellino,[3] and Almerico Murli[1,2]

[1] *University of Naples Federico II, Complesso Universitario M.S. Angelo, Via Cintia, 80126 Naples, Italy*
[2] *SPACI (Southern Partnership of Advanced Computing Infrastructures), c/o Complesso Universitario M.S. Angelo, Via Cintia, 80126 Naples, Italy*
[3] *University of Naples Parthenope, Centro Direzionale, Isola C4, 80143 Naples, Italy*

Correspondence should be addressed to Luisa D'Amore, luisa.damore@unina.it

Received 1 April 2011; Revised 16 June 2011; Accepted 24 June 2011

Academic Editor: Khaled Z. Abd-Elmoniem

In this paper, we consider nonlinear partial differential equations (PDEs) of diffusion/advection type underlying most problems in image analysis. As case study, we address the segmentation of medical structures. We perform a comparative study of numerical algorithms arising from using the semi-implicit and the fully implicit discretization schemes. Comparison criteria take into account both the accuracy and the efficiency of the algorithms. As measure of accuracy, we consider the Hausdorff distance and the residuals of numerical solvers, while as measure of efficiency we consider convergence history, execution time, speedup, and parallel efficiency. This analysis is carried out in a multicore-based parallel computing environment.

1. Introduction

High-quality images are crucial to accurately diagnose a patient or determine treatment. In addition to requiring the best images possible, safety is a crucial consideration. Many imaging systems use X-rays to provide a view of what is beneath a patient's skin. X-ray radiation levels must be kept at a minimum to protect both patients and staff. As a result, raw image data can be extremely noisy. In order to provide clear images, algorithms designed to reduce noise are used to process the raw data and extract the image data while eliminating the noise. In video imaging applications, data often have to be processed at rates of 30 images per second or more. Filtering noisy input data and delivering clear, high-resolution images at these rates require tremendous computing power. This gave rise to the need of developing high-end computing algorithms for image processing and analysis which are able to exploit the high performance of advanced computing machines.

In this paper, we focus on the computational kernels which arise as basic building blocks of the numerical solution of medical imaging applications described in terms of partial differential equations (PDEs) of parabolic/hyperbolic type. Such PDEs arise from the scale-space approach for description of most inverse problems in imaging [1]. One of the main reasons for using PDEs to describe image processing applications is that PDE models preserve the intrinsic locality of many image processing operations. Moreover, we can rely on standard and up-to-date literature and software about basic computational issues arising in such case (such as the construction of suitable discretization schemes, the availability of a range of algorithmic options, and the reuse of software libraries that allow the effective exploitation of high-performance computing resources). Finally, PDEs appear to be effectively implemented on advanced computing environments [2].

We consider two standard discretization schemes of nonlinear time-dependent PDEs: semi-implicit scheme and fully implicit scheme [3]. The former leads to the solution of a linear system at each time (scale) step, while the computational kernel of the fully implicit scheme is the solution of a nonlinear system, to be performed at each time (scale) step. Taking into account that we aim to solve such problems on parallel computer in a scalable way, in the first case, we

use, as linear solver, Krylov iterative methods (GMRES) with algebraic multigrid preconditioners (AMG) [4, 5]. Regarding the fully implicit scheme, we use the Jacobian-Free Newton-Krylov (JFNK) method as nonlinear solver [6].

In recent years, multicore processors are becoming dominant systems in high-performance computing [7]. We provide a multicore implementation of numerical algorithms arising from using the semi-implicit and the implicit discretization schemes of nonlinear diffusion models underlying most problems in image analysis. Our implementation is based on parallel PETSc (Portable Extensible Toolkit for Scientific Computation) computing environment [8]. Parallel software uses a distributed memory model where the details of intercore communications and data managements are hidden within the PETSc parallel objects.

The paper is organized as follows. In Section 2, an overview of the PDE model equation used in describing some of inverse problems in imaging applications will be given. Then, the segmentation problem of medical structures is discussed. Numerical approach will be introduced in Section 3. Section 4 is devoted to the discussion of numerical algorithms based on semi-implicit and implicit numerical schemes. In Section 6, we describe the experiments that we carried out to show both the accuracy and the performance of these algorithms, while Section 7 concludes the work.

2. Diffusion Models Arising in Medical Imaging

The task in medical imaging is to provide in a noninvasive way information about the internal structure of the human body. The basic principle is that the patient is scanned by applying some sort of radiation and its interaction with the body is measured. This result is the data, whose origin has to be identified. Hence, we face an inverse problem. Most medical imaging problems lead to ill-posed (inverse) problems in the sense of Hadamard [9–11]. A standard approach for dealing with such intrinsic instability is to use additional information to construct families of approximate solution. This principle characterizes regularization methods that, starting from the milestone Tikhonov regularization [12], are now one of the most powerful tools for solution of inverse ill-posed problems. In 1992, Rudin et al. introduced the first nonquadratic regularization functional (i.e., the total variation regularization) [13] to denoise images. Moreover, the authors derive the Euler-Lagrange equations as a time-dependent PDE. In the same years Perona and Malik introduced the first nonlinear multiscale analysis [14].

Scale-space theory has been developed by the computer vision community to handle the multiscale nature of image data. A main argument behind its construction is that if no prior information is available about what are the appropriate scales for a given data set, then the only reasonable approach for a vision system is to represent the input data at multiple scales. This means that the original image $u(\mathbf{x}), \mathbf{x} \in \Re^2$ should be embedded into a one-parameter family of derived images, in which fine-scale structures are successively suppressed:

$$SS_\tau : \tau \in \Re \longrightarrow u(\mathbf{x}, \tau). \tag{1}$$

A crucial requirement is that structures at coarse scales in the multiscale representation should constitute simplifications of corresponding structures at finer scales—they should not be accidental phenomena created by the method for suppressing fine-scale structures. A main result is that if rather general conditions are imposed on the types of computations that are to be performed, then convolution by the Gaussian kernel and its derivatives is singled out as a canonical class of smoothing transformations [15, 16].

A strong relation between regularization approaches and the scale-space approach exists via the Euler-Lagrange equation of regularization functionals: it consists of a PDE of parabolic/hyperbolic (diffusion/advection) type [17], defined as follows.

Nonlinear Diffusion Models. Let $\mathbf{x} = (x, y) \in \Re^2$ and $u(\mathbf{x}, \tau)$, defined in $[0, T] \times \Omega$, be the scale-space representation of the brightness function image $u(\mathbf{x})$ defined in $\Omega \subset \Re^2$ describing the real (and unknown) object and $u_0(\mathbf{x})$ the observed image (the input data). Let us consider the following PDE problem:

$$\begin{gathered}\frac{\partial u(\mathbf{x}, \tau)}{\partial \tau} = |\nabla u| \nabla \cdot \left(g(u) \frac{\nabla u}{|\nabla u|} \right) \quad \tau \in [0, T],\ (x, y) \in \Omega \\ u(0, \mathbf{x}) = u_0(\mathbf{x}) \quad \tau = 0,\ (x, y) \in \Omega.\end{gathered} \tag{2}$$

$[0, T]$ is the scale (time) interval; $g(v)$ is a nonincreasing real valued function (for $v > 0$) which tends to zero as $v \to \infty$. Initial and boundary conditions will be provided according to the problem to be solved (denoising, segmentation, deblurring, registration, and so on).

Equations in (2) describe the motion of a curve (a moving front) with a speed depending on a local curvature. Such equations, known as level set equations, were first introduced in [18]. The original idea behind the level set method was a simple one. Given an interface Γ in $\Re^n$ of codimension one (i.e., its dimension is $n - 1$), bounding an (perhaps multiply connected) open region Ω, we wish to analyze and compute its subsequent motion under a velocity field $\vec{v}$. This velocity can depend on position, time, the geometry of the interface (e.g., its normal or its mean curvature), and the external physics. The idea, as devised in 1988 by Osher and Sethian, is merely to define a smooth (at least Lipschitz continuous) function $\phi(x, t)$, that represents the interface Γ as the set where $\phi(x, t) = 0$. Thus, the interface is to be captured for all later time, by merely locating the set $\Gamma(t)$ for which ϕ vanishes. The motion is analyzed by convecting the ϕ values (levels) with the velocity field v. This elementary equation is

$$\frac{\partial \phi}{\partial t} + v \cdot \nabla \phi = 0. \tag{3}$$

Actually, only the normal component of v is needed:

$$v_N = v \cdot \frac{\nabla \phi}{|\nabla \phi|}, \tag{4}$$

and the motion equation becomes

$$\frac{\partial \phi}{\partial t} + v_N \cdot |\nabla \phi| = 0. \tag{5}$$

Taking into account that the mean curvature of $\Gamma(t)$ is

$$\text{cur} = -\nabla \cdot \left(\frac{\nabla \phi}{|\nabla \phi|} \right), \tag{6}$$

equation (5) describes the motion of $\Gamma(t)$ under a speed v_N proportional to its curvature cur (Mean Curvature Motion, MCM equation) [18–20]. This basic model has received a lot of attention because of its geometrical interpretation. Indeed, the level sets of the image solution or level surfaces in 3D images move in the normal direction with a speed proportional to their mean curvature. In image processing, equations like (5) arise in nonlinear filtration, edge detection, image enhancement, and so forth, when we are dealing with geometrical features of the image-like silhouette of object corresponding to level line of image intensity function. Finally, the level set approach instead of explicitly following the moving interface itself takes the original interface Γ and embeds it in higher dimensional scalar function u, defined over the entire image domain. The interface Γ is now represented implicitly as the zeroth level set (or contour) of this function, which varies with space and time (scale) using the partial differential equation in (2), containing terms that are either hyperbolic or parabolic. The theoretical study of the PDE was done by [21] which proved existence and uniqueness of viscosity solutions.

2.1. A Case Study: Image Segmentation. In this paper, we use equations (2) for image segmentation. The task of image segmentation is to find a collection of nonoverlapping subregions of a given image. In medical imaging, for example, one might want to segment the tumor or the white matter of a brain from a given MRI image.

The idea behind level set (also known implicit active contours, or implicit deformable models) for image segmentation is quite simple. The user specifies an initial guess for the contour, which is then moved by image-driven forces to the boundaries of the desired objects. More precisely, the input to the model is a user-defined point-of-view u_0, centered in the object we are interested in segmenting. The output is the function $u(\mathbf{x}, \tau)$. Function $u(\mathbf{x}, \tau)$ in (2) is the *segmentation function,* u_0 represents the initial *contour (initial state of the segmentation function)*, and the image to segment is I^0. Moreover, as proposed in [22], instead of following evolution of a particular level set of u, the PDE model follows the evolution of the entire surface of u under speed law dependent on the image gradient, without regard to any particular level set. Suitably chosen, this flow sharpens the surface around the edges and connects segmented boundaries across the missing information. In [22, 23], the authors formalized such model as the *Riemannian mean curvature flow* where the variability in the parameter ϵ also improves the segmentation process and provides a sort of regularization. Thus, (2) becomes

$$\begin{aligned} \frac{\partial u(\mathbf{x}, \tau)}{\partial \tau} &= \sqrt{\epsilon^2 + |\nabla u|^2} \nabla \cdot \left(g(|\nabla I^0|) \frac{\nabla u}{\sqrt{\epsilon^2 + |\nabla u|^2}} \right) \\ &\qquad \tau \in [0, T],\ \mathbf{x} = (x, y) \in \Omega \\ u(\mathbf{x}, 0) &= u_0(\mathbf{x}) \quad \tau = 0,\ \mathbf{x} = (x,\ y) \in \Omega \\ u(\tau,\ \mathbf{x}) &= 0 \quad \tau \in [0, T],\ \mathbf{x} = (x, y) \in \partial\Omega \end{aligned} \tag{7}$$

accompanied with initial condition u_0 and zero Dirichlet boundary conditions. Regarding u_0, it is usually defined as a circle completely enclosed inside the region that one wish to segment.

The term $g(v)$, called *edge detector*, is a nonincreasing real function such that $g(v) \to 0$ while $v \to \infty$, and it is used for the enhancement of the edges. Indeed, it controls the speed of the diffusion/regularization: if ∇u has a small mean in a neighborhood of a point $\mathbf{x}$, this point $\mathbf{x}$ is considered the interior point of a smooth region of the image and the diffusion is therefore strong. If ∇u has a large mean value on the neighborhood of $\mathbf{x}$, $\mathbf{x}$ is considered an edge point and the diffusion spread is lowered, since $g(v)$ is small for large v. A popular choice in nonlinear diffusion models is the Perona and Malik function [14]: $g(v) = 1/(1 + v^2/\beta)$, $\beta > 0$. In many models, the function $g(|\nabla u|)$ is replaced by its smoothed version $g(|\nabla G_\sigma * u|)$, where G_σ is a smoothing kernel, for example, the Gauss function, which is used in presmoothing of image gradients by the convolution. For shortening notations, we will use abbreviation

$$g = g(|\nabla G_\sigma * u|). \tag{8}$$

In conclusion, we use the *Riemannian mean curvature flow*, as model equation of the segmentation of medical structures: given I_0, the initial image and u_0 equals to a circle contained inside an object of the image I_0, we are interested in segmenting, we compute $u(\mathbf{x}, \tau)$ by solving (7). The level sets of $u(\mathbf{x}, \tau)$, at steady state, provide approximations of the contour to detect.

3. Numerical Schemes

Nonlinear PDE in (7) can be expressed in a compact way as

$$\frac{\partial u(x, y, \tau)}{\partial \tau} = F[u(x, y, \tau, \nabla u(x, y, \tau), I_0)], \tag{9}$$

where

$$F = \sqrt{\epsilon^2 + |\nabla u|^2} \nabla \cdot \left(g(|\nabla I^0|) \frac{\nabla u}{\sqrt{\epsilon^2 + |\nabla u|^2}} \right). \tag{10}$$

Scale Discretization. That is discretization with respect to τ. If $[0, T]$ is the scale interval and *nscales* is the number of scale steps, we denote by τ_i the ith scale-step for all $i = 1, \ldots, nscales$, so that $\tau_{i+1} = \tau_i + \Delta\tau$, where $\Delta\tau = T/nscales$ is the step-size.

Using the Euler forward finite difference scheme to discretize the scale derivative on the left hand side of (9), we get

$$\frac{u(x, y, \tau_i) - u(x, y, \tau_{i-1})}{\Delta\tau} = F[u(x, y, \tau)] \tag{11}$$

or, equivalently

$$u(x, y, \tau_i) = u(x, y, \tau_{i-1}) + \Delta\tau \cdot F[u(x, y, \tau)]. \tag{12}$$

Let us denote as $u_i = u(x, y, \tau_i)$, $i = 1, \ldots, nscales$, the function u evaluated at τ_i. Equation (12) is rewritten as

$$u_i = u_{i-1} + \Delta\tau \cdot F(u(x, y, \tau)) \equiv G[u(x, y, \tau)]. \tag{13}$$

Depending on the collocation value, used to evaluate $u(x, y, \tau)$ with respect to the parameter τ, inside the F function on the right hand side of (12) three iterative schemes derive:

(i) *explicit scheme*: $u_i = G[u_{i-1}]$, that is, the function F is evaluated at $u_{i-1} = u(x, y, \tau_{i-1})$;

(ii) *semi-implicit scheme*: $u_i = G[u_{i-1}, u_i]$, that is, we use u_i to discretize the numerator $|\nabla u|$ of the fraction $\nabla u/\sqrt{\epsilon^2 + |\nabla u|^2}$. Other quantities are evaluated at u_{i-1};

(iii) *implicit scheme*: $u_i - G[u_{(i)}] = 0$, that is, the function F is evaluated at u_i.

In summary, the difference between the semi-implicit and the implicit scheme relies on the scale discretization of the term $|\nabla u|$ at the numerator of $\nabla u/\sqrt{\epsilon^2 + |\nabla u|^2}$ inside the function F. This term controls the diffusion process, and it plays the role of *edge-enhancement*. If we consider the three-dimensional (3D) domain $\Omega_T = \Omega \times [0, T]$, the semi-implicit scheme employs a sort of 2D + 1 discretization of Ω_T proceeding along *nscales* two dimensional (2D) slices each one obtained at $\tau \equiv \tau_i$, while the fully implicit scheme uses a fully 3D discretization of Ω_T. This difference suggests that the fully implicit scheme may provide a more accurate edge detection than the semi-implicit scheme. This difference is highlighted by considering their discretization errors.

Space Discretization. That is discretization with respect to (x, y). If Ω is the space domain, we introduce a rectangular uniform grid on Ω consisting of $N_x \times N_y$ (for simplicity we assume that Ω is a rectangular of dimension 1×1; this means that $h_x = 1/N_x$ and $h_y = 1/N_y$), nodes $(x_i, y_j) = (l\Delta x, m\Delta y)$,$l = 1, \ldots, N_x$, $m = 1, \ldots, N_y$, and we use finite volumes to discretize the partial derivatives of u, as in [24, 25].

Scale-Space Discretization. Let

$$u_i^{l,m} = u(x_l, y_m, \tau_i) \in \Re^{N_x \times N_y \times nscales} \tag{14}$$

be the vector obtained from the scale-space discretization of the function u, we have the following iteration formulas.

(i) Explicit scheme:

$$\frac{u_i^{l,m} - u_{i-1}^{l,m}}{\Delta\tau} = \sqrt{\epsilon^2 + \left|\nabla u_{i-1}^{l,m}\right|^2}\nabla \cdot \left(g(|\nabla I^0|)\frac{\nabla u_{i-1}^{l,m}}{\sqrt{\epsilon^2 + \left|\nabla u_{i-1}^{l,m}\right|^2}}\right) \Longleftrightarrow \tag{15}$$
$$u_i^{l,m} = \left(I + \Delta\tau[A]_{i-1}^{l,m}\right)u_{i-1}^{l,m} \quad \forall i = 1, 2, \ldots, N_E,$$

where, for each i, the matrix $[A]_{i-1}^{l,m} \in \Re^{N_x^2 \times N_y^2}$ and $I \in \Re^{N_x^2 \times N_y^2}$ is the unit matrix, while N_E is the scale steps number.

(i) Semi-implicit scheme:

$$\frac{u_i^{l,m} - u_{i-1}^{l,m}}{\Delta\tau} = \sqrt{\epsilon^2 + \left|\nabla u_{i-1}^{l,m}\right|^2}\nabla \cdot \left(g(|\nabla I^0|)\frac{\nabla u_i^{l,m}}{\sqrt{\epsilon^2 + \left|\nabla u_{i-1}^{l,m}\right|^2}}\right) \Longleftrightarrow$$
$$u_i^{l,m} = u_{i-1}^{l,m} + \Delta\tau[A]_{i-1}^{l,m}u_i^{l,m} \Longleftrightarrow$$
$$\left(I + \Delta\tau[A]_{i-1}^{l,m}\right)u_i^{l,m} = u_{i-1}^{l,m} \quad \forall i, i = 1, \ldots, N_{\mathrm{SI}} \tag{16}$$

where, for each i, the matrix $[A]_{i-1}^{l,m} \in \Re^{N_x^2 \times N_y^2}$ and $I \in \Re^{N_x^2 \times N_y^2}$ is the unit matrix and N_{SI} is the scale steps number.

(ii) Fully-implicit scheme:

$$\frac{u_i^{l,m} - u_{i-1}^{l,m}}{\Delta\tau} = \sqrt{\epsilon^2 + \left|\nabla u_i^{l,m}\right|^2}\nabla \cdot \left(g(|\nabla I^0|)\frac{\nabla u_i^{l,m}}{\sqrt{\epsilon^2 + \left|\nabla u_i^{l,m}\right|^2}}\right) \Longleftrightarrow \tag{17}$$
$$u_i^{l,m} = u_{i-1}^{l,m} + \Delta\tau[A]_i^{l,m}\left(u_i^{l,m}\right) \quad \forall i = 1, \ldots, N_I,$$

where N_I is the scale steps number and $[A]_i^{l,m}$, for each i, is a nonlinear vector operator on $\Re^{N_x^2 \times N_y^2}$, depending on $u_i^{l,m}$.

In particular, we apply the Crank-Nicholson scheme [3] which uses the average of the forward Euler method at step $i-1$ and the backward Euler method at step i:

$$\begin{aligned}\frac{u_i^{l,m}-u_{i-1}^{l,m}}{\Delta\tau} &= \frac{1}{2}\left[\sqrt{\epsilon^2+\left|\nabla u_i^{l,m}\right|^2}\nabla\right.\\ &\quad\left.\cdot\left(g(|\nabla I^0|)\frac{\nabla u_i^{l,m}}{\sqrt{\epsilon^2+\left|\nabla u_i^{l,m}\right|^2}}\right)\right]+\frac{1}{2}\cdots\\ &\quad\cdots\left[\sqrt{\epsilon^2+\left|\nabla u_{i-1}^{l,m}\right|^2}\nabla\right.\\ &\quad\left.\cdot\left(g(|\nabla I^0|)\frac{\nabla u_{i-1}^{l,m}}{\sqrt{\epsilon^2+\left|\nabla u_{i-1}^{l,m}\right|^2}}\right)\right],\\ &\qquad\forall i=1,\ldots,N_I\\ \Longleftrightarrow\ u_i^{l,m} &= u_{i-1}^{l,m}+\Delta\tau\left[B_{i-1}^{l,m}u_{i-1}^{l,m}+[A]_i^{l,m}\left(u_i^{l,m}\right)\right],\\ &\qquad\forall i=1,\ldots,\mathrm{N}_I,\end{aligned}\tag{18}$$

where $B_{i-1}^{l,m}$ is a matrix on $\Re^{N_x^2\times N_y^2}$, depending on $u_{i-1}^{l,m}$, and $[A]_i^{l,m}$ is a nonlinear vector operator on $\Re^{N_x^2\times N_y^2}$, depending on $u_i^{l,m}$.

4. Algorithms and Their Computational Complexity

The effectiveness of these schemes depends on a suitable balance between accuracy (scale-space discretization error), number of flop/s per iteration (algorithm complexity), and the total execution time needed to reach a prescribed accuracy (software performance).

Let us denote the discretization error E_d. It is

$$E_d = O\left(h_x^p\right)+O\left(h_y^p\right)+O(\Delta\tau^q).\tag{19}$$

Explicit scheme is accurate at the first order both with respect to scale and space, that is, $p=q=1$; anyway, it is the one straightforwardly computable. The computational kernel is a matrix-vector product, at every scale step. This scheme requires very small time steps in order to be stable (CFL (Courant-Friedrich-Levy) condition that guarantees the stability of the evolution), and its use is limited rather by its stability than accuracy. This constraint is practically very restrictive, since it typically leads to the need for a huge amount of iterations [3]. Semi-implicit scheme is absolutely stable for all scale steps. The accuracy, in terms of discretization error with respect to both scale and space, is of the first order, because $p=1$, $q=2$ [24, 25]. Crank-Nicholson provides a discretization error of second order, that is, $p=q=2$, but it requires extra computations, leading to a nonlinear system of equations, at every time step, while stability is ensured for all scale steps [3]. In the following, we collect these results:

$$\begin{aligned}&\text{explicit scheme: } p=q=1,\\ &\text{semi-implicit scheme: } p=1,\ q=2,\\ &\text{implicit scheme: } p=q=2.\end{aligned}\tag{20}$$

Then, the fully implicit scheme provides an order of accuracy greater than that provided by the others. This difference may be important in those applications of image analysis where the edges are fundamental to recognize some pathologies.

Algorithm complexity of these schemes depends on the choice of the numerical solver. Concerning the semi-implicit scheme, we employ Krylov subspace methods, which are the most effective approaches for solving large linear systems [5]. In particular, we use Generalized Minimal RESidual method (GMRES) equipped with Algebraic multigrid (AMG) preconditioner. Such techniques are convenient because they require as input only the system matrix corresponding to the finest grid. In addition, they are suitable to implement in a parallel computing environment. For the fully implicit scheme we use the Jacobian Free Newton Krylov Method (JFNK) [6]. JFNK methods are synergistic combinations of Newton-type methods for superlinearly convergent solution of nonlinear equations and Krylov subspace methods for solving the Newton correction equations. The link between the two methods is the Jacobian-vector product, which may be probed approximately without forming and storing the elements of the true Jacobian.

Let us briefly describe the numerical algorithms that we are going to implement, which are based on the semi-implicit and the implicit discretization schemes, together with their complexity.

Algorithm SI (Semi-Implicit Scheme). For all $i=1,\ldots,N_{\mathrm{SI}}$ solution of

$$\left(I+\Delta\tau[A]_{i-1}^{l,m}\right)u_i^{l,m}=u_{i-1}^{l,m}\Longleftrightarrow HS_{i-1}^{l,m}u_i^{l,m}=u_{i-1}^{l,m},\tag{21}$$

with respect to $u_i^{l,m}$. $HS_{i-1}^{l,m}$, for each i, is a matrix $\in\Re^{N_x^2\times N_y^2}$. As space derivative we use the 2nd order finite covolume discretization scheme (see [24, 25] for convergence, consistence and stability). By this way, matrix $[A]_{i-1}^{l,m}$ is a block pentadiagonal matrix with tridiagonal blocks along the main diagonal and diagonal blocks along the upper and lower diagonals.

Algorithm I (Implicit Scheme). For all $i=1,\ldots,N_I$ solution of

$$\begin{aligned}u_i^{l,m} &= u_{i-1}^{l,m}+\Delta\tau\left[B_{i-1}^{l,m}u_{i-1}^{l,m}+[A]_i^{l,m}\left(u_i^{l,m}\right)\right]\\ &\Longleftrightarrow HI_i^{l,m}\left(u_i^{l,m},u_{i-1}^{l,m}\right)=0,\end{aligned}\tag{22}$$

with respect to $u_i^{l,m}$. $HI_i^{l,m}$, for each i, is a nonlinear vector operator on $\Re^{N_x^2\times N_y^2}$.

Compute $r_0 = P(b - Ax_0)$, $\beta := \|r_0\|_2$ and $v_1 := r_0/\beta$
Define the $(m+1) \times mH_k = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$. Set $H_m = 0$.
for $k = 1$ to m do
  Compute $w_j := PAv_j$
  For $i = 1$ to k do:
  $h_{ij} := (w_j, v_i)$
  $w_j := w_j - h_{ij}v_i$
end for
  $h_{j+1,j} = \|w_j\|_2$. If $h_{j+1,j} = 0$ set $m := j$ and go to 12
  $v_{j+1} = w_j/h_{j+1,j}$
end for
Compute y_m the minimiser of $\|\beta e_1 - H_m y\|_2$
Set $x_m := x_0 + V_m y_m$.

Algorithm 1: Preconditioned GMRES for solving a linear system $Ax = b$. Input: A (matrix coefficient), b (right hand side), P (preconditioner). Output: x_m, approximate solution at the mth step. For all iteration, a matrix-vector product is required.

Algorithm SI. For each scale step, to solve the linear system (21), we employ GMRES iterative method (see Algorithm 1). Computational kernel of GMRES is a matrix-vector product. Taking into account the structure of the coefficient matrix (we assume that $N_x = N_y = N$, then $h = 1/N = h_x = h_y$), the computational cost of GMRES is

$$T_{\text{GMRES}}(N^2) = O\left(k_{\text{GMRES}}^{\text{SI}} \cdot 5N^2\right), \tag{23}$$

where $k_{\text{GMRES}}^{\text{SI}}$ is the maximum iterations of GMRES (over the scale steps). Computational complexity of Algorithm SI is

$$T_{\text{SI-GMRES}}(N^2) = O\left(N_{\text{SI}} \cdot k_{\text{GMRES}}^{\text{SI}} \cdot 5N^2\right). \tag{24}$$

Algebraic multigrid (AMG) method follows the main idea of (geometric) multigrid (MG), where a sequence of grids is constructed from the underlying geometry with corresponding transfer operators between the grids [26]. The main idea of MG is to remove the smooth error, that cannot be eliminated by relaxation on the fine grid, by coarse-grid correction. The solution process then as usual consists of presmoothing, transfer of residuals from fine to coarse grids, interpolation of corrections from coarse to fine levels, and optional postsmoothing. In contrast to geometric multigrid, the idea of AMG is to define an artificial sequence of systems of equations decreasing in size. We call these equations *coarse-grid* equations. The interpolation operator $P_{lv}^{l,m}$ and the restriction operator $R_{lv}^{l,m}$ define the transfer from finer to coarser grids and vice versa. Finally, the operator on the coarser grid at level $lv + 1$ is defined by

$$A_{lv+1}^{l,m} = R_{lv}^{l,m} A_{lv}^{l,m} P_{lv}^{l,m}. \tag{25}$$

The AMG method consists of two main parts, the setup phase and the solution phase. During the setup phase, the coarse-grids and the corresponding operators are defined. The solution phase consists of a multilevel iteration. The number of recursive calls, which is the number of levels lv, depends on the size and structure of the matrix. For our case, we use the V-cycle pattern with the FALGOUT-CLJP coarse grid selection [27]. Looking at the Algorithm SI in Table 1, the preconditioner P is just A_{lv+1}.

Table 1: Comparisons between two algorithms: $E_d = O(10^{-1})$.

Algorithm	$\Delta\tau$	Number of scale steps	d_H	Execution time (secs)
SI	0.16	3	0.9492	9.262
I	0.4	1	0.9498	7.675

Computational cost of each iteration of GMRES is that of the AMG preconditioner plus the matrix-vector products:

$$T_{\text{AMG+GMRES}}(N^2) = O\left(k_{\text{GMRES}}^{\text{SI}}\left(\underbrace{T_{\text{AMG}}(lv)}_{lv \cdot N^2} + 5N^2\right)\right), \tag{26}$$

then, we get:

$$T_{\text{SI+AMG+GMRES}}(N^2) = O\left(N_{\text{SI}} k_{\text{GMRES}}^{\text{SI}} lv N^2\right). \tag{27}$$

Following picture shows a schematic description of Algorithm SI that emphasizes its main steps and the most time consuming operation, that is, the matrix vector products needed at each step of GMRES.

Algorithm I. For each scale step, to solve the nonlinear equations (22), we employ the Jacobian-Free Newton-Krylov (JFNK) method. JFNK is a nested iteration method consisting of at least two and usually four levels. The primary levels, which give the method its name, are the loop over the Newton method:

$$HI_i^{l,m}\left(u_{n+1}^{l,m}\right) = 0 \iff HI_{\text{i}}^{l,m}\left(u_n^{l,m}\right) + J_i\left(u_n^{l,m}\right)\left(u_{n+1}^{l,m} - u_n^{l,m}\right) = 0, \tag{28}$$

and the loop building up the Krylov subspace out of which each Newton step is drawn:

$$J_i\left(u_n^{l,m}\right)\delta u_n^{l,m} = -HI_i^{l,m}\left(u_n^{l,m}\right), \qquad u_{n+1}^{l,m} = u_n^{l,m} + \delta u_n^{l,m}, \tag{29}$$

```
for i = 1 to N_SI do
   for k = 1 to k_GMRES do
      for lv = 1 to levels_AMG do
         matrix-vector products involving HS_i^{l,m}, A_{lv+1}^{l,m} and u_i^{l,m}(k, lv)
end for
```

Algorithm 2: Sketch of Algorithm SI.

```
for i = 1 to N_I do
   for n = 1 to N_New do
      for k = 1 to k_GMRES do
         evaluate HI_i^{l,m}, u_i^{l,m}(n, k)
end for
```

Algorithm 3: Sketch of Algorithm I.

Outside of the Newton loop, a globalization method is often required. We use line search.

Forming each element of J which is a matrix of dimension $N^2 \times N^2$ requires taking derivatives of the system of equations with respect to u. This can be time consuming. Using the first order Taylor series expansion of $HI_i^{l,m}(u_n^{l,m} + \rho v)$, it follows that

$$J_i\left(u_n^{l,m}\right)\delta u_n^{l,m} = \frac{\left[HI_i^{l,m}\left(u_n^{l,m} + \rho\delta u_n^{l,m}\right) - HI_i^{l,m}\left(u_n^{l,m}\right)\right]}{\rho} + O(\rho^2), \tag{30}$$

where ρ is a perturbation. JFNK does not require the formation of the Jacobian matrix; it instead forms a result vector that approximates this matrix multiplied by a vector. This *Jacobian-free* approach has the advantage to provide the quadratic convergence of Newton method without the costs of computing and storing the Jacobian. Conditions are provided on the size of ρ that guarantee local convergence.

Algorithm 3 shows a schematic description of Algoritm I that emphasizes its main steps and the most time consuming operation, that is, evaluations of the nonlinear operator $HI_i^{l,m}$ at each innermost step.

Algorithm complexity of JFNK is

$$T_{\text{JFNK}}(N^2) = O\left(N_{\text{New}} k^I_{\text{GMRES}}[f + O(N^2)]\right), \tag{31}$$

where N_{New} is the maximum number of Newton's steps, over the scale steps, k^I_{GMRES} is the maximum number of GMRES iterations (computed over Newton's steps and scale steps)), f is the number of evaluations of $HI_i^{l,m}$. Finally, we get

$$T_{I+\text{JFNK}}(N^2) = O\left(N_I N_{\text{New}} k^I_{\text{GMRES}}[f + O(N^2)]\right). \tag{32}$$

A straightforward comparison between the algorithm complexity of these algorithms shows that Algorithm SI asymptotically seems to be comparable with respect to Algorithm SI. Of course, the performance analysis must also take into account the efficiency of these two schemes in a given computing environment. Next section describes the PETSc-based implementation of these algorithms that we have developed in a multicore computing environment.

5. The Multicore-Based Implementation

The software has been developed using the high-level software library PETSc (Portable Extensible Toolkit for Scientific Computations) (release 3.1, March 2010) [8]. PETSc provides a suite of data structures and routines as building blocks for the implementation of large-scale codes to be used in scientific applications modeled by partial differential equation. PETSc is flexible: its modules, that can be used in application codes written in Fortran, C, and C++, are developed by means of object-oriented programming techniques.

The library has a hierarchical structure: it relies on standard basic computational (BLAS, LAPACK) and communication (MPI) kernels and provides mechanism needed to write parallel application codes. PETSc transparently handles the moving of data between processes without requiring the user to call any data transfer function. This includes handling parallel data layouts, communicating ghost points, gather, scatter and broadcast operations. Such operations are optimized to minimize synchronization overheads.

Our parallelization strategy is based on domain decomposition: in particular, we adopt the *row-block data distribution*, which is the standard PETSc data distribution. Row-block data distribution means that blocks of dimensions $N^2/p \times N$ of contiguous rows of matrices of dimension $N^2 \times N^2$ are distributed among contiguous processes. By the same way, vectors of size N are distributed among p processors as blocks of size N/P. Such partitioning has been chosen because overheads, due to redistribution before the solution of the linear systems, are avoided. Further, row-block data distribution introduces a coarse grain parallelism which is best oriented to exploit concurrency of multicore multiprocessors because it does not require a strong cooperation among computing elements: each computing element has to locally manage the blocks that are assigned to it.

The computing platform that we consider is made of 16 blades (1 blade consisting of 2 quad core Intel Xeon E5410@2.33 GHz) Dell PowerEdge M600, equipped with IEEE double precision arithmetic. Because high performance technologies can be employed in medical applications only to the extent that the overall cost of the infrastructure is

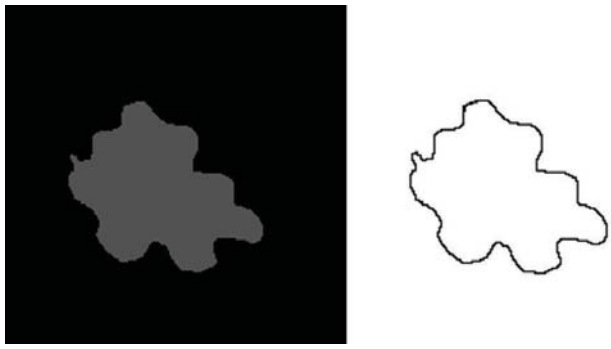


Figure 1: Test 1. Image size: 840 × 840, simulated image and its contour to compute.

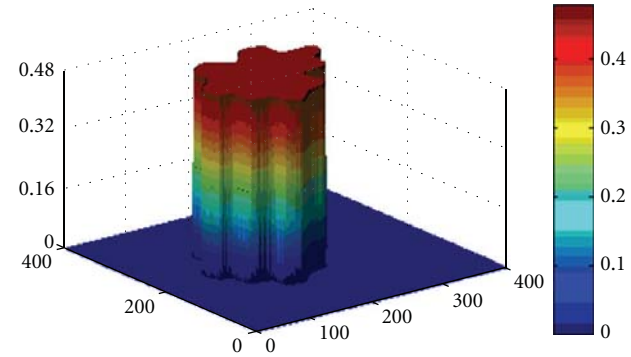


Figure 2: Test 1: The 3D visualization of the segmentation function $u(\tau, \mathbf{x})$ at steady state $T = 0.4$.

affordable, and because we consider single images of medium size, we show results obtained by using 1 blade, that is, we run the parallel algorithms on up to $p = 8$ cores of a single blade. Of course, in case of multiple images or sequences of images, the use of a greater number of cores may be interesting.

6. Experiments

In this section we present and discuss computational results obtained by implementing Algorithm I and Algorithm SI in a multicore parallel computing machine. Before illustrating experimental results, let us briefly describe the choice of

(1) test images,

(2) comparison criteria,

(3) parameters selection.

(1) Test Images. we have carried out many experiments in order to analyze the performance of these algorithms. Here we show results concerning the segmentation of a (malignant) melanoma (see Figure 7) [28]. Epiluminescence microscopy (ELM) has proven to be an important tool in the early recognition of malignant melanoma [29, 30]. In ELM, halogen light is projected onto the object, thus rendering the surface translucent and making subsurface structures visible. As an initial step, the mask of the skin lesion is determined by a segmentation algorithm. Then, a set of features containing shape and radiometric features as well as local and global parameters is calculated to describe the malignancy of the lesion. In order to better validate computed results and to analyze the software performance, we first consider a synthetic test image simulating the object we are interested in segmenting (see Figure 1).

(2) Comparison Criteria. We compare the algorithms using the following criteria:

(a) distance from original solution. As measure of the difference between two curves, we use the Hausdorff distance measured between the computed curve and the original one. It is well known that the Hausdorff distance is a metric over the set of all closed bounded sets (see [31]), here we restrict ourselves to finite point sets because that is all that is necessary for segmentation algorithms [32]. Given two finite point sets C_1 and C_2, the Hausdorff distance d_H between the sets C_1 and C_2 is defined as follows:

$$d_H(C_1, C_2) = \max\{h(C_1, C_2), h(C_2, C_1)\},$$
$$h(C_1, C_2) = \max_{a \in C_1} \min_{b \in C_2} \|a - b\|, \tag{33}$$

where $\|\cdot\|$ is the euclidean norm. It identifies the point $a \in C_1$ that is fastest from any point of C_2 and vice versa, then it keeps the maximum,

(b) efficiency: execution time of (serial) algorithms,

(c) convergence history: behavior of residuals and iteration numbers of inner solvers,

(d) Parallel performance: execution time, speedup, and efficiency versus cores number.

(3) Parameters Selection. We set $K = 1.0$ and $\epsilon = 1.0$. Let us explain how we select the values of the scale step size and the number of scale steps. Regarding $\Delta\tau$, its value is chosen according to that required to E_d. Taking into account that, in Algorithm SI, E_d is accurate at the first order with respect to $\Delta\tau_{\mathrm{SI}}$, while it is accurate at the second order with respect to $\Delta\tau_I$, in Algorithm I, by requiring that the discretization error is about the same, we get

$$E_d = O(\Delta\tau_{\mathrm{SI}}) = O(\Delta\tau_I^2) \Longrightarrow \Delta\tau_{\mathrm{SI}} = \sqrt{\Delta\tau_I}. \tag{34}$$

Finally, in Algorithm SI, the stopping criterion of the linear solver (GMRES) uses the tolerance

$$\mathrm{TOL} = 10^{-10}, \tag{35}$$

while the preconditioner AMG uses the tolerance $\mathrm{TOL} = 10^{-7}$ and 25 as maximum number of AMG-levels. In the Algorithm 2, the stopping criterion of the nonlinear solver uses the tolerance

$$\mathrm{TOL} = 10^{-10}. \tag{36}$$

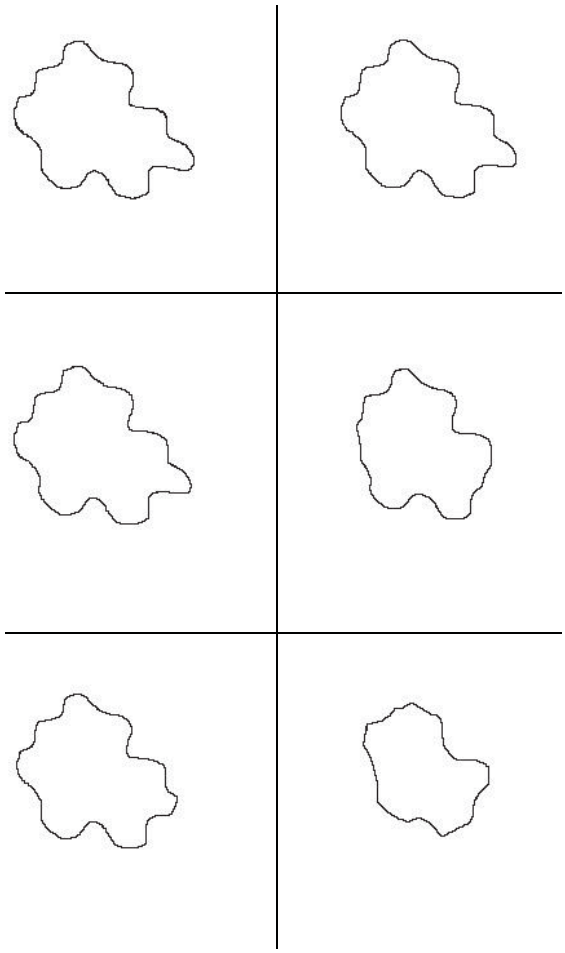


Figure 3: Test 1: Comparisons between segmentation results. On the left Algorithm I. On the right Algorithm SI. First row: $\Delta\tau_I = 0.4$, $\Delta\tau_{\mathrm{SI}} = 0.16$. Second row: $\Delta\tau_I = 0.04$, $\Delta\tau_{\mathrm{SI}} = 0.16$. Third row: $\Delta\tau_I = 0.004$, $\Delta\tau_I = 0.0016$.

Regarding the number of scale steps (N_{SI} and N_I), taking into account that

$$N_{\mathrm{SI},I} = \frac{T}{\Delta\tau_{\mathrm{SI},I}}, \tag{37}$$

its choice depends on $\Delta\tau_{\mathrm{SI},I}$ and on the value of $\tau \equiv T$, that is, the value of the scale parameter corresponding to steady state of the segmentation function $u(\mathbf{x}, \tau)$, solution of the PDE model. To check the steady state, we require that the residuals, corresponding to different scale steps, reach the tolerance

$$\mathrm{TOL} = 10^{-9}. \tag{38}$$

We found that this corresponds to $T = 0.4$ (see Figure 2) for Test 1 and to $T = 2$ for Test 2.

Test 1: Synthetic Image. In Tables 1, 2, and 3, we show the Hausdorff distance and execution time by requiring that discretization error is of the first, second, and third order, that is, $E_d = O(10^{-1})$, $E_d = O(10^{-2})$, and $E_d = O(10^{-3})$,

Table 2: Comparisons between two algorithms: $E_d = O(10^{-2})$.

Algorithm	$\Delta\tau$	Number of scale steps	d_H	Execution time (secs)
SI	0.016	25	0.6412	61.487
I	0.04	10	0.9498	71.67

Table 3: Comparisons between two algorithms: $E_d = O(10^{-3})$.

Algorithm	$\Delta\tau$	Number of scale steps	d_H	Execution time (secs)
SI	0.0016	250	0.5993	356.926
I	0.004	100	0.9492	356.335

respectively. Hence, we get the following values of the scale step size:

$$\begin{aligned} \Delta\tau_I = 0.4 &\Longrightarrow \Delta\tau_{\mathrm{SI}} = 0.16, \\ \Delta\tau_I = 0.04 &\Longrightarrow \Delta\tau_{\mathrm{SI}} = 0.016, \\ \Delta\tau_I = 0.004 &\Longrightarrow \Delta\tau_{\mathrm{SI}} = 0.0016. \end{aligned} \tag{39}$$

Moreover, concerning the number of scale steps, it follows that

$$\begin{aligned} E_d = O(10^{-1}) &\Longrightarrow N_I = \frac{0.4}{0.4} = 1, \quad N_{\mathrm{SI}} = \mathrm{int}\left[\frac{0.4}{0.16}\right] = 3, \\ E_d = O(10^{-2}) &\Longrightarrow N_{\mathrm{SI}} = \frac{0.4}{0.04} = 10, \quad N_{\mathrm{SI}} = \frac{0.4}{0.016} = 25, \\ E_d = O(10^{-3}) &\Longrightarrow N_{\mathrm{SI}} = \frac{0.4}{0.04} = 100, \quad N_{\mathrm{SI}} = \frac{0.4}{0.016} = 250. \end{aligned} \tag{40}$$

Note that while in the first case, that is, if we require $E_d = O(10^{-1})$, these two algorithms are quite numerically equivalent, both in terms of execution time and of the computed result; as discretization error decreases, Algorithm I appears to be more robust than Algorithm SI, in the sense that Algorithm I reaches the steady state with high accuracy (the Hausdorff distance is of 95%), while the computed results of Algorithm SI are less accurate, even though the execution time of Algorithm I sometimes slightly increases. These results suggest that if it needs to get an accurate and reliable result, Algorithm I should be preferable. Figure 3 show segmentation results. Finally, note that the execution time of these two algorithms asymptotically is the same, as stated by the analysis of computational cost carried on in Section 4.

Convergence History. Convergence history is illustrated by showing the behavior of relative residuals versus the scale steps (see Figures 4, 5, and 6), and by reporting iteration number of the GMRES and of Newton's method, respectively, (see Tables 4, 5, 6, and 7). We consider $\Delta\tau_{\mathrm{SI}} = 0.16$, 0.016, and $\Delta\tau_I = 0.04$.

In the following, we show results corresponding to the segmentation of a melanoma (see Figure 7). As expected, because this is a real image, the steady state is reached at a

Table 4: Convergence history of Algorithm SI. $\Delta\tau_{\mathrm{SI}} = 0.16$. First column reports the scale step number, second column reports the number of GMRES iterations at each scale step, and last column reports the maximum number of AMG levels at each GMRES iteration.

i	k^i_{gmres}	lv^i
1	8	6
2	6	5
3	5	3

Table 5: Convergence history of Algorithm SI. $\Delta\tau_{\mathrm{SI}} = 0.016$. On the first column, we denote the scale step number, and the symbol $p - q$ is used to denote the steps ranging from the p-th until to the q-th. Second column reports the number of GMRES iterations at each scale step, third column reports the maximum number of AMG levels at each GMRES iteration.

i	k^i_{gmres}	lv^i
1	5	6
2-3	5	5
4	5	4
5–7	4	4
8-8	4	3
10–13	3	3
14–20	3	2
21-22	3	1
23–25	2	1

Table 6: Convergence history of Algorithm SI. $\Delta\tau_I = 0.4$. First column reports the scale step number, second column reports the number of Newton's steps at each scale step, and last column reports the number of GMRES iterations at each Newton's step.

i	N^i_{New}	k^i_{GMRES}
1	10	9, 9, 8, 8, 8, 7, 7, 7, 6, 6

Table 7: Convergence history of Algorithm SI. $\Delta\tau_I = 0.04$. First column reports the scale step number, second column reports the number of Newton's steps at each scale step, and last column reports the number of GMRES iterations at each Newton's step.

i	N^i_{New}	k^i_{GMRES}
1	10	9, 9, 8, 8, 7, 7, 7, 7, 6, 6
2	10	9, 8, 8, 8, 7, 7, 7, 6, 6, 6
3	9	9, 8, 8, 7, 7, 6, 6, 6, 6
4	9	8, 8, 8, 7, 7, 6, 6, 6, 5
5	9	8, 8, 7, 7, 7, 6, 6, 5, 5
6	7	8, 7, 7, 7, 6, 5, 5
7	7	7, 7, 7, 7, 6, 5, 5
8	6	7, 7, 6, 6, 6, 5
9	6	7, 7, 6, 6, 5, 5
10	5	7, 6, 6, 5, 4

scale greater than that of the synthetic test image, that is, $T = 2$, thus both algorithms require a greater number of scale steps to the reach the steady state. Tables 8, 9, and 10, and Figure 8, compare results of Algorithm SI and Algorithm I.

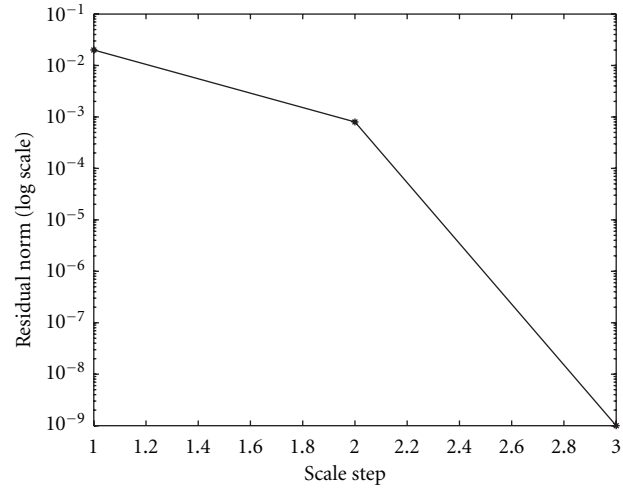


Figure 4: Algorithm SI. Behavior of the relative residual $\|r_i\|_2/\|r_0\|_2$ versus 3 scale steps. $\Delta\tau_{\mathrm{SI}} = 0.16$.

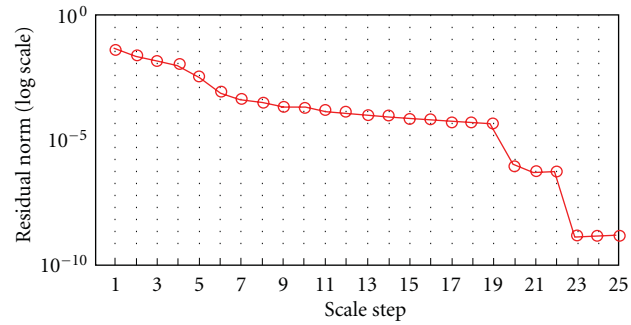


Figure 5: Algorithm SI. Behavior of the relative residual $\|r_i\|_2/\|r_0\|_2$ versus 25 scale steps. $\Delta\tau_{\mathrm{SI}} = 0.016$.

Table 8: Test 2: Comparisons between two algorithms: $E_d = O(10^{-1})$.

Algorithm	$\Delta\tau$	Number of scale steps	Execution time (secs)
SI	0.16	13	41.125
I	0.4	5	47.893

Table 9: Test 2: Comparisons between two algorithms: $E_d = O(10^{-2})$.

Algorithm	$\Delta\tau$	Number of scale steps	Execution time (secs)
SI	0.016	125	417.56
I	0.04	50	455.33

Table 10: Test 2: Comparisons between two algorithms: $E_d = O(10^{-3})$.

Algorithm	$\Delta\tau$	Number of scale steps	Execution time (secs)
SI	0.0016	1250	4828.5
I	0.004	500	4663.7

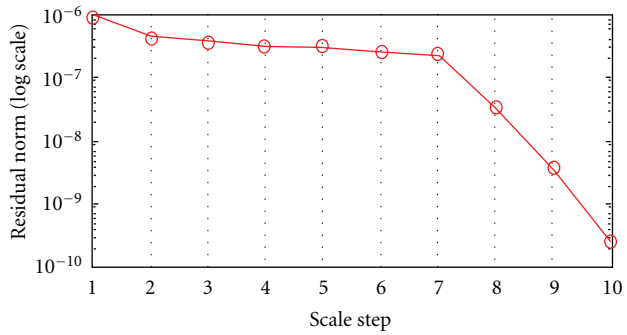


FIGURE 6: Algorithm I. Behavior of the relative residual $\|r_i\|_2/\|r_0\|_2$ versus 10 scale steps. $\Delta\tau_I = 0.04$.

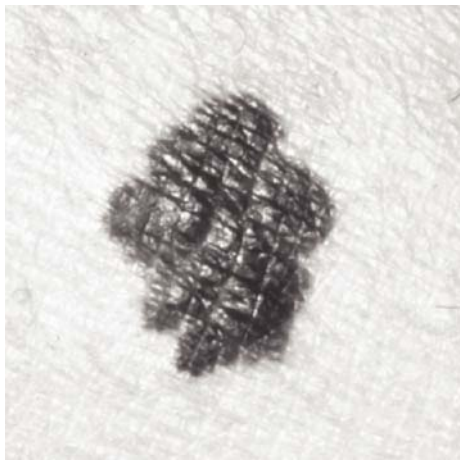


FIGURE 7: Test 2: ELM of a melanoma. Image size is 840×840.

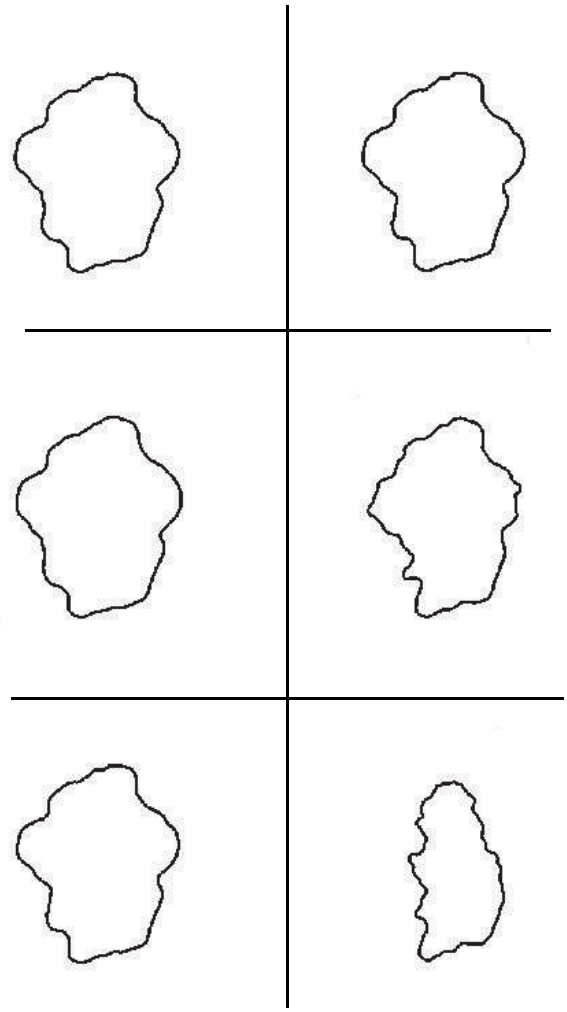


FIGURE 8: Test 2: Comparisons between segmentation results. On the left Algorithm I. On the right Algorithm SI. First row:, $\Delta\tau_I = 0.4$, $\Delta\tau_{\mathrm{SI}} = 0.16$. Second row: $\Delta\tau_I = 0.04$, $\Delta\tau_{\mathrm{SI}} = 0.16$. Third row: $\Delta\tau_I = 0.004$, $\Delta\tau_I = 0.0016$.

Parallel Performance. We show the performance of the multicore-based parallel algorithms and their scalability as the number of cores increases. We run the parallel algorithms using up to $p = 8$ cores of the parallel machine.

Following Figures report execution time, speedup, and efficiency of Algorithm SI, at scale step size $\Delta\tau = 0.16$ (i.e., $E_d = (10^{-1})$ (i.e., Figures 9, 10, and 11), then same results are shown at scale step size $\Delta\tau = 0.016$, corresponding to $E_d = O(10^{-2})$ (i.e., Figures 12, 13, and 14).

Finally, we report execution time, speedup, and efficiency of Algorithm I, at scale step $\Delta\tau = 0.4$ (i.e., Figures 15, 16, and 17) and $\Delta\tau = 0.04$ (i.e., Figures 18, 19, and 20), respectively. Note that parallel efficiency of both algorithms always is, at least, of 60% and, on average, of about 80%. In particular, parallel efficiency of Algorithm I is about 90%. Execution time of both algorithms reduces to about 2 seconds on eight cores in the first case (i.e., $E_d = (10^{-1})$, and to about 10 seconds in the second case ($E_d = O(10^{-2})$). This means that, in a multicore computing environment, both algorithms provide the requested solution within a response time that can be considered quite acceptable in medical imaging applications and, in particular, that Algorithm I is competitive with Algorithm SI.

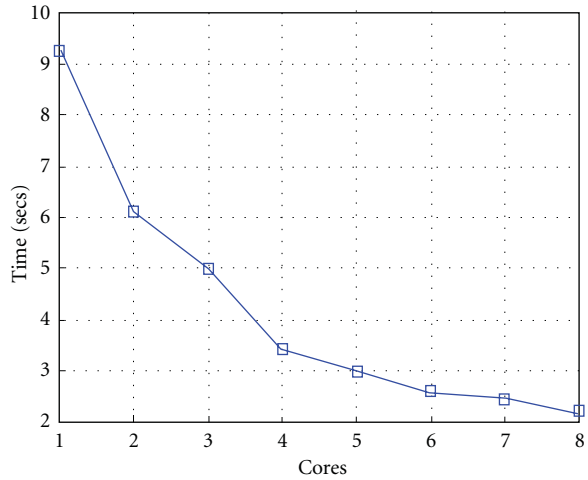


FIGURE 9: Test 1: Algorithm SI: Total execution time versus the number of cores. $\Delta\tau = 0.16$. 3 scale steps.

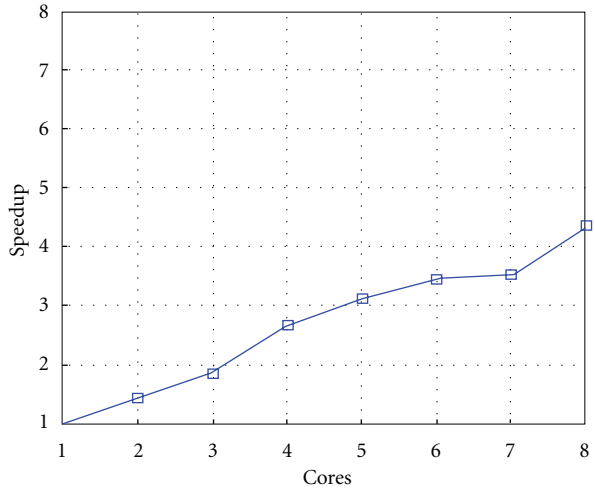


Figure 10: Test 1: Algorithm SI: Speedup versus the number of cores. $\Delta\tau = 0.16$. 3 scale steps.

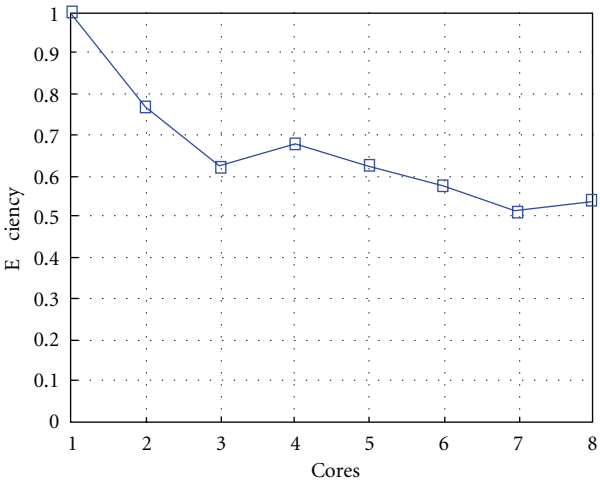


Figure 11: Test 1: Algorithm SI: Parallel efficiency versus the number of cores. $\Delta\tau = 0.16$. 3 scale steps.

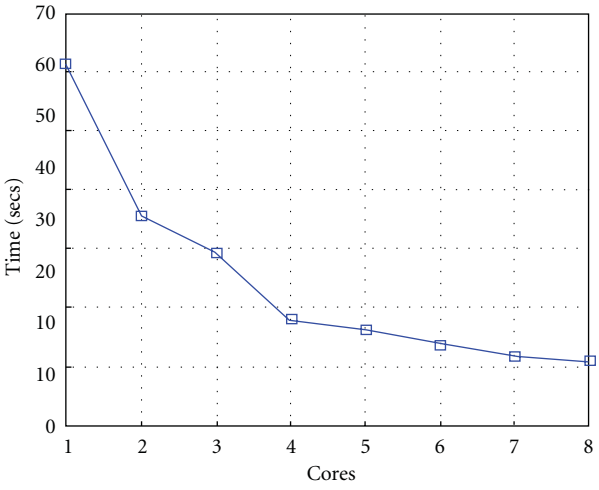


Figure 12: Test 1: Algorithm SI: Total execution time versus the number of cores. $\Delta\tau = 0.016$.

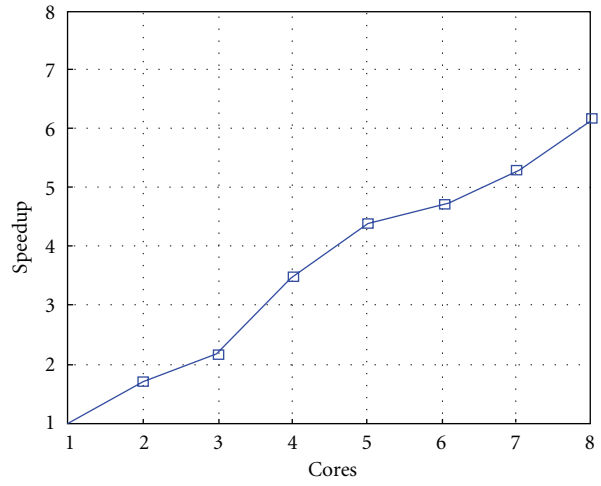


Figure 13: Test 1: Algorithm SI: Speedup versus the number of cores. $\Delta\tau = 0.016$.

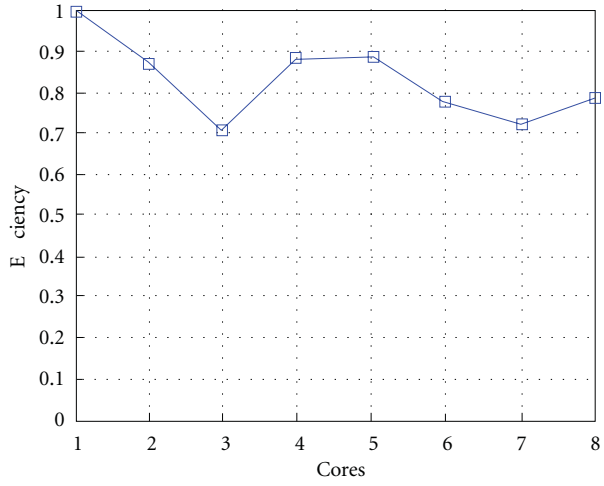


Figure 14: Test 1: Algorithm SI: Parallel efficiency versus the number of cores. $\Delta\tau = 0.016$.

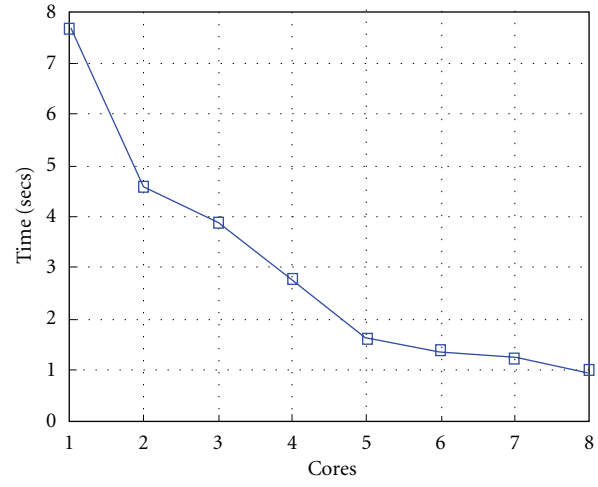


Figure 15: Test 1: Algorithm I: Total execution time versus the number of cores. $\Delta\tau_I = 0.4$.

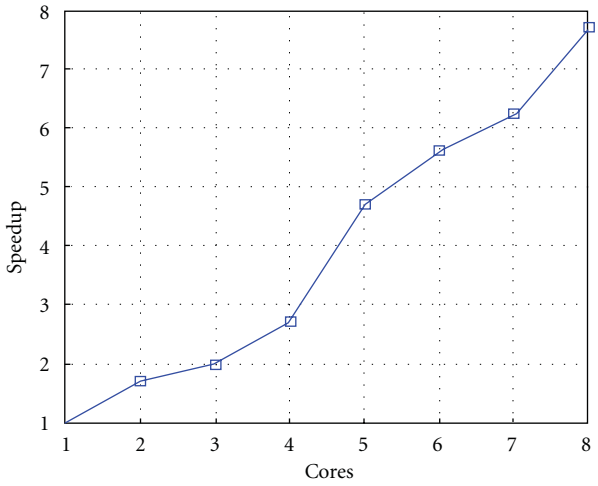


Figure 16: Test 1: Algorithm I: Speedup. $\Delta\tau_I = 0.4$.

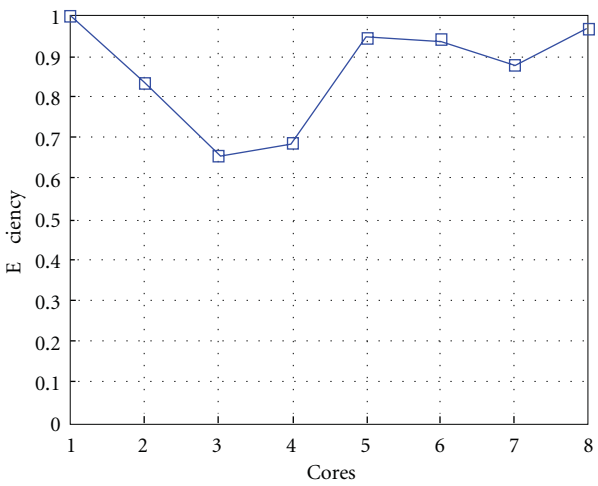


Figure 17: Test 1: Algorithm I: Efficiency. $\Delta\tau_I = 0.4$.

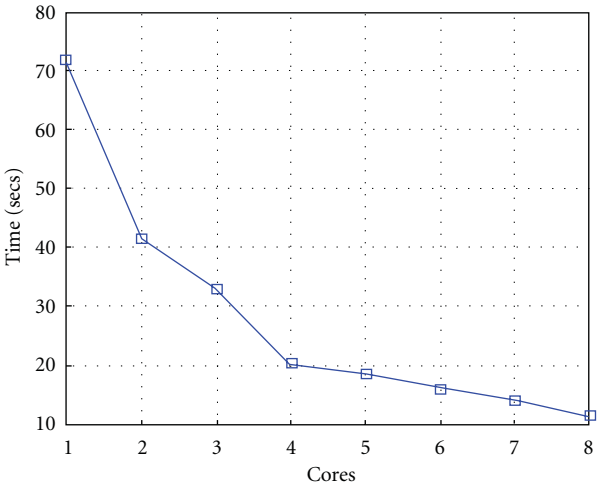


Figure 18: Test 1: Algorithm I: Total execution time versus the number of cores. $\Delta\tau_I = 0.04$.

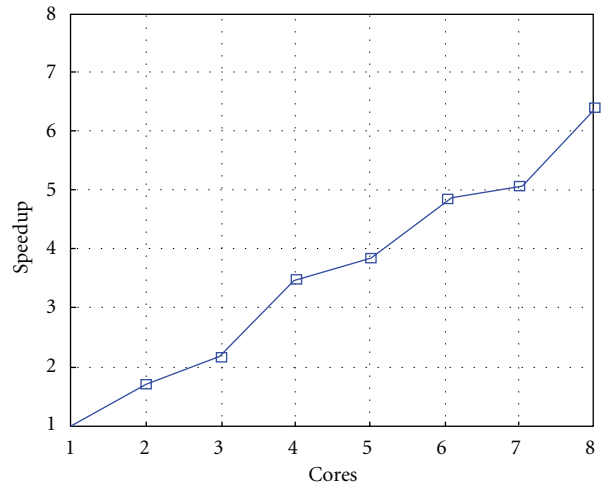


Figure 19: Test 1: Algorithm I: Speedup. $\Delta\tau_I = 0.04$.

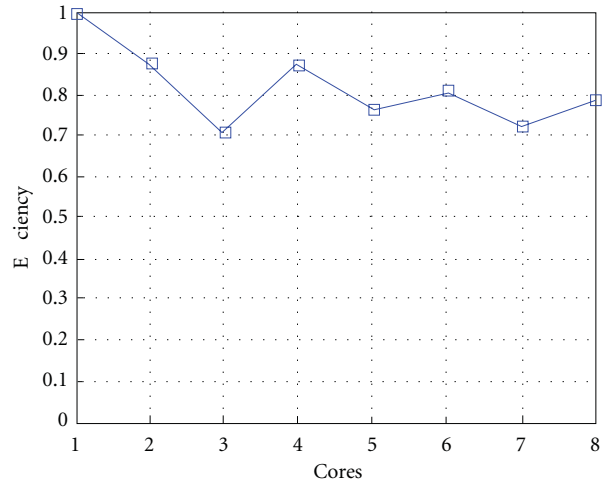


Figure 20: Test 1: Algorithm I: Efficiency. $\Delta\tau_I = 0.04$.

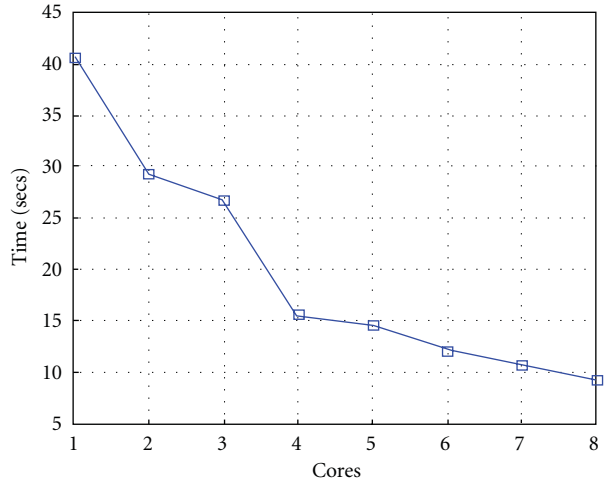


Figure 21: Test 2: Algorithm SI: Total execution time versus the number of cores. $\Delta\tau_I = 0.16$.

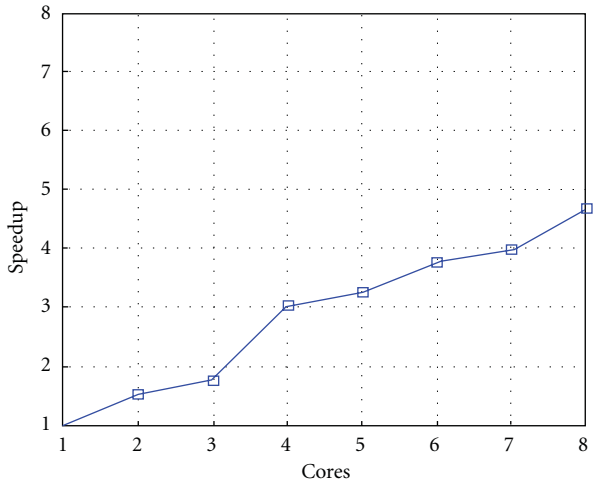


Figure 22: Test 2: Algorithm SI: Speedup. $\Delta\tau_I = 0.16$.

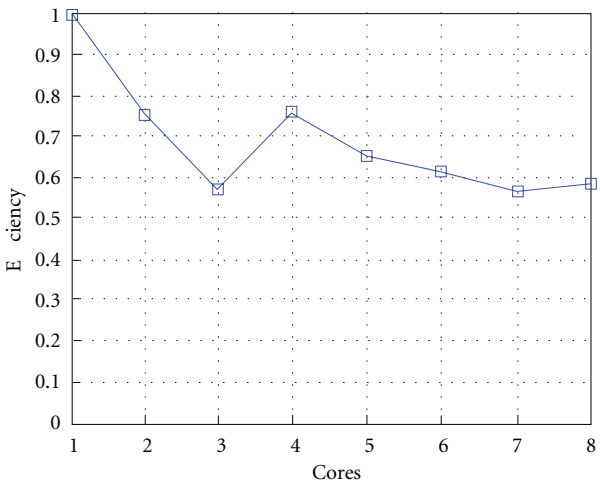


Figure 23: Test 2: Algorithm SI: Efficiency. $\Delta\tau_I = 0.16$.

Figures 21, 22, 23, 24, 25, and 26 show time, speedup, and parallel efficiency of two algorithms in case of Test 2. We only consider the case of $\Delta\tau_{\mathrm{SI}} = 0.16$ and $\Delta\tau_I = 0.4$.

Finally, we show results on scalability of parallel algorithms. Let $T_p(N)$ be the execution time of the parallel algorithm running on p cores for segmenting an image of size $N^2 \times N^2$. We measure the scalability of these algorithms by measuring $T_p(N)$ as N varies, once p is fixed, and by measuring $T_p(N)$ as N and p grow. We note that, in case of Algorithm SI, the scaling factor is

$$\frac{T_p(2N)}{T_p(N)} \simeq 4.2, \tag{41}$$

while, for Algorithm I, we get as scaling factor

$$\frac{T_p(2N)}{T_p(N)} \simeq 2.3. \tag{42}$$

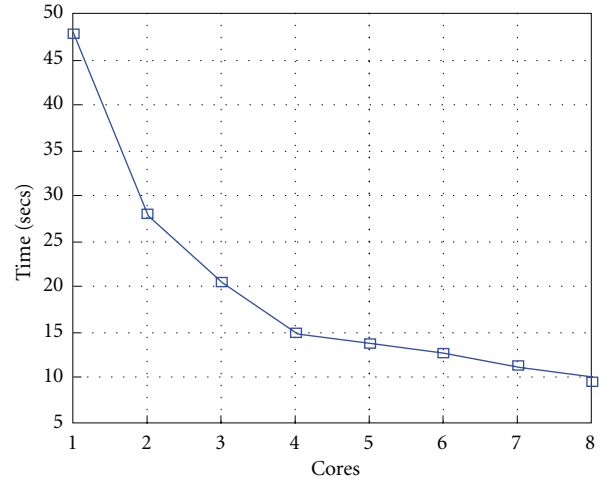


Figure 24: Test 2: Algorithm I: Total execution time versus the number of cores. $\Delta\tau_I = 0.4$.

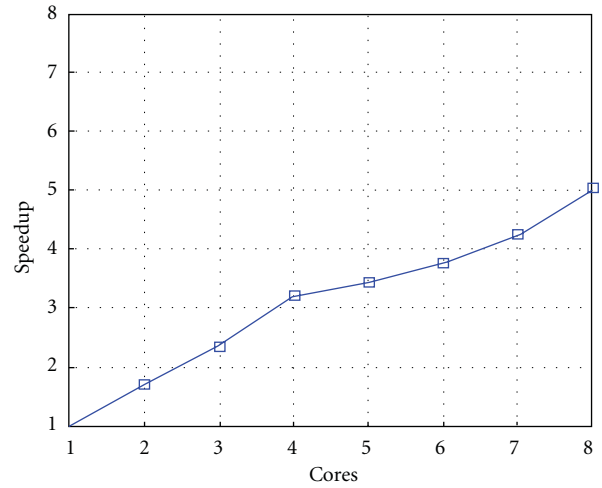


Figure 25: Test 2: Algorithm I. Speed up. $\Delta\tau_I = 0.4$.

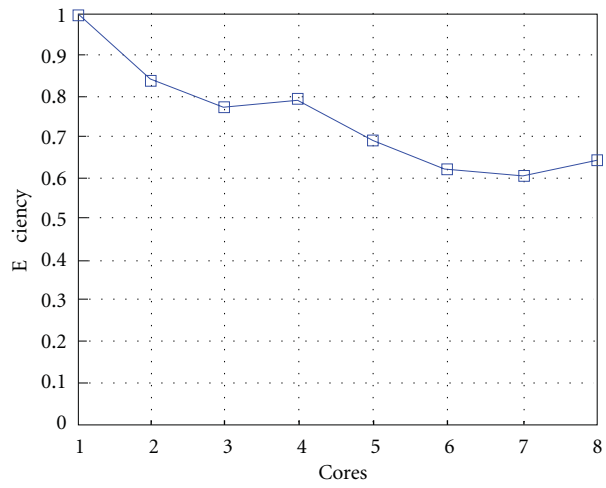


Figure 26: Test 2: Algorithm I. Efficiency. $\Delta\tau_I = 0.4$.

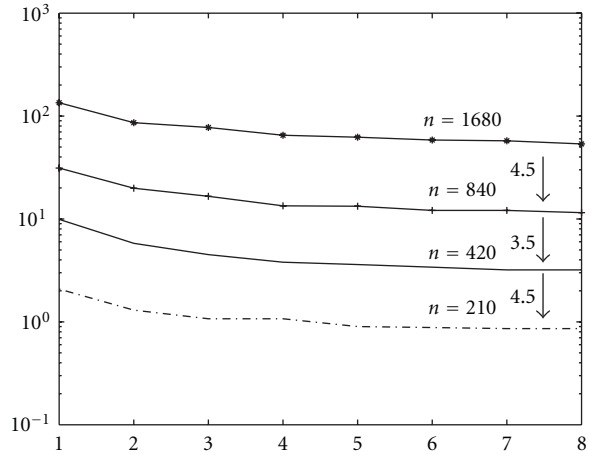


Figure 27: Scalability of parallel Algorithm SI, as N is fixed and p varies, and $N = 210, 420, 840, 1680$. Each line refers to the execution time of the algorithm at a fixed value of N.

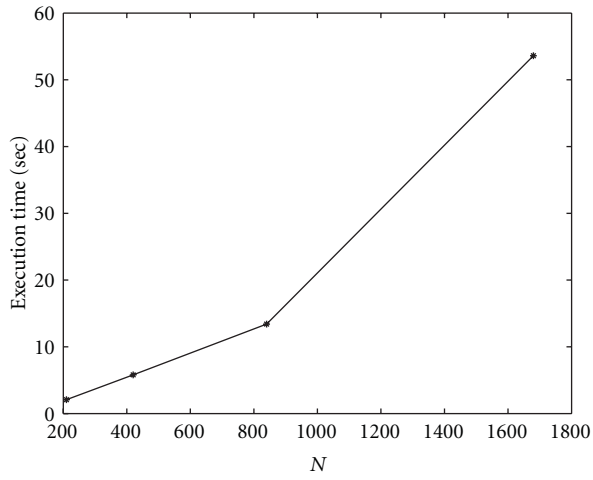


Figure 28: Scalability of parallel Algorithm SI, as N and p vary. Each point of the graph refers to the execution time of the parallel semi-implicit algorithm at ($p = k \cdot p_1$, $N = k \cdot N_1$), where $p_1 = 1$, $N_1 = 210$, and $k = 1, 2, 3, 4$.

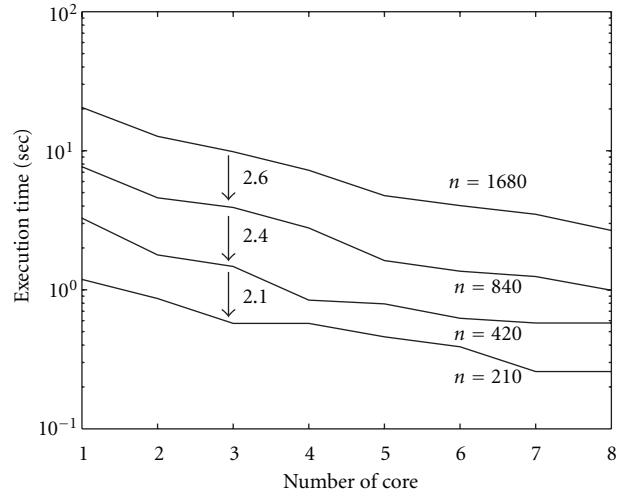


Figure 29: Scalability of parallel Algorithm I, as N is fixed and p varies, and $N = 210, 420, 840, 1680$. Each line refers to the execution time of the algorithm at a fixed value of N. $\Delta\tau_I = 0.4$.

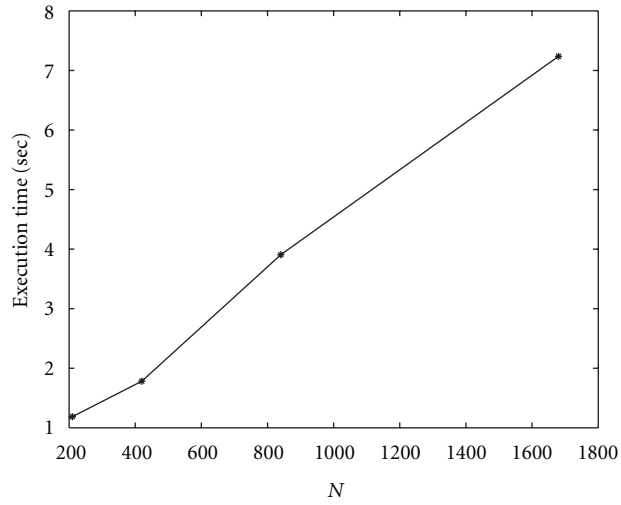


Figure 30: Scalability of parallel Algorithm I, as N and p vary. Each point of the graph refers to the execution time of the parallel implicit algorithm at ($p = k \cdot p_1$, $N = k \cdot N_1$), where $p_1 = 1$, $N_1 = 210$ and $k = 1, 2, 3, 4$. $\Delta\tau_I = 0.4$.

This means that Algorithm I scales better than Algorithm SI. In Figures 27 and 28 (for Algorithm SI) and Figure 29 and 30 (for Algorithm SI), we report $T_p(N)$ as N and p varies. In particular, each point of the graph refers to the execution time of the parallel algorithm at ($p = k \cdot p_1$, $N = k \cdot N_1$), where $p_1 = 1$, $N_1 = 210$, and $k = 1, 2, 3, 4$.

7. Conclusions

A straightforward comparison between the semi-implicit and the fully implicit discretization schemes of nonlinear PDE of parabolic/hyperbolic type states that fully implicit discretization usually leads to too expensive algorithms. In this paper, we provide a multicore implementation of two numerical algorithms arising from using these two discretization schemes: semi-implicit (Algorithm SI) and fully implicit (Algorithm I). Taking into account that we aim to solve such problems on parallel computer in a scalable way, in the first case, we use, as linear solver, Krylov iterative methods (GMRES) with algebraic multigrid preconditioners (AMG). Regarding the fully implicit scheme, we use the Jacobian-Free-Newton Krylov (JFNK) method as nonlinear solver.

We compare these two algorithms using different metrics measuring both the accuracy and the efficiency. We note that if we require that the discretization error E_d is $E_d = O(10^{-1})$, these two algorithms are quite numerically equivalent, both in terms of execution time and of the computed result; while, as discretization error decreases, Algorithm I appears to be more robust than Algorithm SI, in the sense that

Algorithm I reaches the steady state with high accuracy (the Hausdorff distance is of 95%), while the computed results of Algorithm SI are less accurate, even though the execution time of Algorithm I sometimes slightly increases. These results suggest that if it needs to get accurate and reliable results, Algorithm I should be preferable.

The parallel efficiency of both algorithms always is, at least, of 60% and, on average, of about 80%. In particular, parallel efficiency of Algorithm I is of about 90%. Execution time of both algorithms reduces to about 2 seconds on eight cores if $E_d = (10^{-1})$ and to about 10 seconds if $E_d = O(10^{-2})$. This means that, in a multicore computing environment, Algorithm I is competitive with Algorithm SI.

In conclusion, our results suggest that if it is required high accuracy of the computed solution in a suitable turnaround time, using a multicore computing environment fully implicit scheme provides an accurate and reliable solution within a response time of few seconds, quite acceptable in medical imaging applications, such as computer-aided-diagnosis.

References

[1] G. Aubert and P. Kornprobst, *Mathematical Problems in Image Processing: Partial Differential Equations and the Calculus of Variations*, vol. 147 of *Applied Mathematical Sciences*, Springer, 2nd edition, 2006.

[2] http://www.cs.purdue.edu/research/cse/pses.

[3] J. W. Thomas, *Numerical Partial Differential Equations*, vol. 22 of *Text in Applied Mathematics*, Springer, 1995.

[4] Y. Saad and M. Schultz, "GMRES: a generelized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM Journal on Scientific Computing*, vol. 7, pp. 856–869, 1986.

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, 2nd edition, 1993.

[6] D. A. Knoll and D. E. Keyes, "Jacobian-free Newton-Krylov methods: a survey of approaches and applications," *Journal of Computational Physics*, vol. 193, no. 2, pp. 357–397, 2004.

[7] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy, "The impact of multicore on computational science software," *CTWatch Quarterly*, vol. 3, no. 1, 2007.

[8] http://www.mcs.anl.gov/petsc/petsc-as/index.html.

[9] M. Bertero and P. Boccacci, *Introduction to Inverse Problems in Imaging*, IOP Publishers, Bristol, UK, 1998.

[10] A. K. Louis, "Medical imaging: state of the art and future development," *Inverse Problems*, vol. 8, no. 5, pp. 709–738, 1992.

[11] W. Rundell and H. Eng, *Inverse Problems in Medical Imaging & Nondesctructive Testing*, Springer, 1997.

[12] A. N. Tikhonov and V. Y. Arsenin, *Solutions of Ill-Posed Problems*, John Wiley & Sons, New York, NY, USA, 1977.

[13] L. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D*, vol. 60, no. 1–4, pp. 259–268, 1992.

[14] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 7, pp. 629–639, 1990.

[15] J. J. Koenderink, "The structure of images," *Biological Cybernetics*, vol. 50, no. 5, pp. 363–370, 1984.

[16] T. Lindeberg, *Scale-Space Theory in Computer Vision*, Springer, 1994.

[17] O. Scherzer and J. Weickert, "Relations between regularization and diffusion filtering," *Journal of Mathematical Imaging and Vision*, vol. 12, no. 1, pp. 43–63, 2000.

[18] S. Osher and J. A. Sethian, "Fronts propagating with curvature-dependent speed: algorithms based on hamilton-jacobi formulations," *Journal of Computational Physics*, vol. 79, no. 1, pp. 12–49, 1988.

[19] J. A. Sethian, *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision and Materials Science*, Cambridge University Press, Cambridge, UK, 1999.

[20] S. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, Springer, 2003.

[21] L. C. Evans and J. Spruck, "Motion of level sets by mean curvature I," *Transaction of the American Mathematical Society*, vol. 330, no. 1, 1992.

[22] A. Sarti and G. Citti, "Subjective surface and Riemannian mean curvature flow graphs," *Acta Mathematica Universitatis Comenianae*, vol. 70, no. 1, pp. 85–104, 2001.

[23] R. Malladi and J. A. Sethian, "Level set methods for curvature flow, image enchancement, and shape recovery in medical images," in *Visualization and Mathematics*, H. C. Hege and K. Polthier, Eds., pp. 329–345, Springer, Heidelberg, Germany, 1997.

[24] N. J. Walkington, "Algorithms for computing motion by mean curvature," *SIAM Journal on Numerical Analysis*, vol. 33, no. 6, pp. 2215–2238, 1996.

[25] A. Handlovivcov, K. Mikula, and F. Sgallari, "Semi-implicit complementary volume scheme for solving level set like equations in image processing and curve evolution," *Numerishe Mathematik*, vol. 93, no. 4, pp. 675–695, 2003.

[26] J. W. Ruge and K. Stüuben, "Algebraic multigrid methods (AMG) applied to fluid flow problems," *Frontiers in Applied Mathematics*, 1986.

[27] V. E. Henson and U. M. Yang, "BoomerAMG: a parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[28] H. Ganster, A. Pinz, R. Röhrer, E. Wildling, M. Binder, and H. Kittler, "Automated melanoma recognition," *IEEE Transactions on Medical Imaging*, vol. 20, no. 3, pp. 233–239, 2001.

[29] H. Pehamberger, A. Steiner, and K. Wolff, "In vivo epiluminescence microscopy of pigmented skin lesions. I. Pattern analysis of pigmented skin lesions," *Journal of the American Academy of Dermatology*, vol. 17, no. 4, pp. 571–583, 1987.

[30] F. Nachbar, W. Stolz, T. Merkle et al., "The ABCD rule of dermatoscopy: high prospective value in the diagnosis of doubtful melanocytic skin lesions," *Journal of the American Academy of Dermatology*, vol. 30, no. 4, pp. 551–559, 1994.

[31] A. Csazar, *General Topology*, Adam HIlger, Bristol, UK, 1978.

[32] D. P. Huttenlocher, G. A. Klanderman, and W. J. Rucklidge, "Comparing images using the hausdorff distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 15, no. 9, pp. 850–863, 1993.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 952819, 16 pages
doi:10.1155/2011/952819

Research Article

True 4D Image Denoising on the GPU

Anders Eklund,[1,2] Mats Andersson,[1,2] and Hans Knutsson[1,2]

[1] *Division of Medical Informatics, Department of Biomedical Engineering, Linköping University, Linköping, Sweden*
[2] *Center for Medical Image Science and Visualization (CMIV), Linköping University, Linköping, Sweden*

Correspondence should be addressed to Anders Eklund, anders.eklund@liu.se

Received 31 March 2011; Revised 23 June 2011; Accepted 24 June 2011

Academic Editor: Khaled Z. Abd-Elmoniem

The use of image denoising techniques is an important part of many medical imaging applications. One common application is to improve the image quality of low-dose (noisy) computed tomography (CT) data. While 3D image denoising previously has been applied to several volumes independently, there has not been much work done on true 4D image denoising, where the algorithm considers several volumes at the same time. The problem with 4D image denoising, compared to 2D and 3D denoising, is that the computational complexity increases exponentially. In this paper we describe a novel algorithm for true 4D image denoising, based on local adaptive filtering, and how to implement it on the graphics processing unit (GPU). The algorithm was applied to a 4D CT heart dataset of the resolution $512 \times 512 \times 445 \times 20$. The result is that the GPU can complete the denoising in about 25 minutes if spatial filtering is used and in about 8 minutes if FFT-based filtering is used. The CPU implementation requires several days of processing time for spatial filtering and about 50 minutes for FFT-based filtering. The short processing time increases the clinical value of true 4D image denoising significantly.

1. Introduction

Image denoising is commonly used in medical imaging in order to help medical doctors to see abnormalities in the images. Image denoising was first applied to 2D images [1–3] and then extended to 3D data [4–6], 3D data can either be collected as several 2D images over time or as one 3D volume. A number of medical imaging modalities (e.g., computed tomography (CT), ultrasound (US) and magnetic resonance imaging (MRI)) now provide the possibility to collect 4D data, that is, time-resolved volume data. This makes it possible to, for example, examine what parts of the brain that are active during a certain task (functional magnetic resonance imaging (fMRI)). While 4D CT data makes it possible to see the heart beat in 3D, the drawback is that a lower amount of X-ray exposure has to be used for 4D CT data collection, compared to 3D CT data collection, in order to not harm the patient. When the amount of exposure is decreased, the amount of noise in the data increases significantly.

Three-dimensional image denoising has previously been applied to several time points independently, but there has not been much work done on *true* 4D image denoising where the algorithm considers several volumes at the same time (and not a single volume at a time). Montagnat et al. [7] applied 4D anisotropic diffusion filtering to ultrasound volumes and Jahanian et al. [8] applied 4D wavelet denoising to diffusion tensor MRI data. For CT data, it can be extra beneficial to use the time dimension in the denoising, as some of the reconstruction artefacts vary with time. It is thereby possible to remove these artefacts by taking full advantage of the 4D data. While true 4D image denoising is very powerful, the drawback is that the processing time increases exponentially with respect to dimensionality.

The rapid development of graphics processing units (GPUs) has resulted in that many algorithms in the medical imaging domain have been implemented on the GPU, in order to save time and to be able to apply more advanced analysis. To give an example of the rapid GPU development, a comparison of three consumer graphic cards from Nvidia is given in Table 1. The time frame between each GPU generation is 2-3 years. Some examples of fields in medical imaging that have taken advantage of the computational power of the GPU are image registration [9–13], image segmentation [14–16] and fMRI analysis [17–20].

In the area of image denoising, some algorithms have also been implemented on the GPU. Already in 2001 Rumpf and

Table 1: Comparison between three Nvidia GPUs, from three different generations, in terms of processor cores, memory bandwidth, size of shared memory, cache memory, and number of registers; MP stands for multiprocessor and GB/s stands for gigabytes per second. For the GTX 580, the user can for each kernel choose to use 48 KB of shared memory and 16 KB of L1 cache or vice versa.

Property/GPU	9800 GT	GTX 285	GTX 580
Number of processor cores	112	240	512
Normal size of global memory	512 MB	1024 MB	1536 MB
Global memory bandwidth	57.6 GB/s	159.0 GB/s	192.4 GB/s
Constant memory	64 KB	64 KB	64 KB
Shared memory per MP	16 KB	16 KB	48/16 KB
Float registers per MP	8192	16384	32768
L1 cache per MP	None	None	16/48 KB
L2 cache	None	None	768 KB

Strzodka [21] described how to apply anisotropic diffusion [3] on the GPU. Howison [22] made a comparison between different GPU implementations of anisotropic diffusion and bilateral filtering for 3D data. Su and Xu [23] in 2010 proposed how to accelerate wavelet-based image denoising by using the GPU. Zhang et al. [24] describe GPU-based image manipulation and enhancement techniques for dynamic volumetric medical image visualization, but enhancement in this case refers to enhancement of the visualization, and not of the 4D data. Recently, the GPU has been used for real-time image denoising. In 2007, Chen et al. [25] used bilateral filtering [26] on the GPU for real-time edge-aware image processing. Fontes et al. [27] in 2011 used the GPU for real-time denoising of ultrasound data and Goossens et al. [28] in 2010 managed to run the commonly used nonlocal means algorithm [29] in real time.

To our knowledge, there has not been any work done about true 4D image denoising on the GPU. In this work we therefore present a novel algorithm, based on local adaptive filtering, for 4D denoising and describe how to implement it on the GPU, in order to decrease the processing time and thereby significantly increase the clinical value.

2. Methods

In this section, the algorithm that is used for true 4D image denoising will be described.

2.1. The Power of Dimensionality. To show how a higher number of dimensions, the power of dimensionality, can improve the denoising result, a small test is first conducted on synthetic data. The size of the 4D data is $127 \times 127 \times 9 \times 9$, but there is no signal variation in the last two dimensions. The data contains a large step, a thin line, and a shading from the top left corner to the bottom right corner. A large amount of 4D additive noise was finally added to the data. Image denoising of different dimensionality was then applied. For the 2D case, the denoising was done on one 127×127 image, for the 3D case, the denoising was done on one $127 \times 127 \times 9$ volume and for the 4D case all the data was used. A single anisotropic lowpass filter was used for the denoising, and the filter had the same dimensionality as the data and was oriented along the structures. The original test data, the test data with noise and the denoising results are given in Figure 1. It is clear that the denoising result is improved significantly for each new dimension.

2.2. Adaptive Filtering in 4D. The denoising approach that our work is based on is adaptive filtering. It was introduced for 2D by Knutsson et al. in 1983 [2] and then extended to 3D in 1992 [4]. In this work, the same basic principles are used for adaptive filtering in 4D. The main idea is to first estimate the local structure tensor [30] (by using a first set of filters) in each neighbourhood of the data and then let the tensor control the reconstruction filters (a second set of filters). The term reconstruction should in this paper not be confused with the reconstruction of the CT data. The local structure tensor $\mathbf{T}$ is in 4D a 4×4 symmetric matrix in each time voxel,

$$\mathbf{T} = \begin{pmatrix} t_1 & t_2 & t_3 & t_4 \\ t_2 & t_5 & t_6 & t_7 \\ t_3 & t_6 & t_8 & t_9 \\ t_4 & t_7 & t_9 & t_{10} \end{pmatrix} = \begin{pmatrix} xx & xy & xz & xt \\ xy & yy & yz & yt \\ xz & yz & zz & zt \\ xt & yt & zt & tt \end{pmatrix}, \quad (1)$$

and contains information about the local structure in the data, that can be used to control the weights of the reconstruction filters. The result of the adaptive filtering is that smoothing is done along structures (such as lines and edges in 2D), but not perpendicular to them.

2.3. Adaptive Filtering Compared to Other Methods for Image Denoising. Compared to more recently developed methods for image denoising (e.g., nonlocal means [29], anisotropic diffusion [3] and bilateral filtering [26]), adaptive filtering is in our case used for 4D image denoising for three reasons. First, adaptive filtering is computationally more efficient than the other methods. Nonlocal means can give very good results, but the algorithm can be extremely time consuming (even if GPUs are used). Anisotropic diffusion is an *iterative* algorithm and can therefore be rather slow. Adaptive filtering is a *direct* method that does not need to be iterated. Bilateral filtering does not only require a multiplication for each filter coefficient and each data value, but also an evaluation of the intensity range function (e.g., an exponential) which is much more expensive to perform than a multiplication.

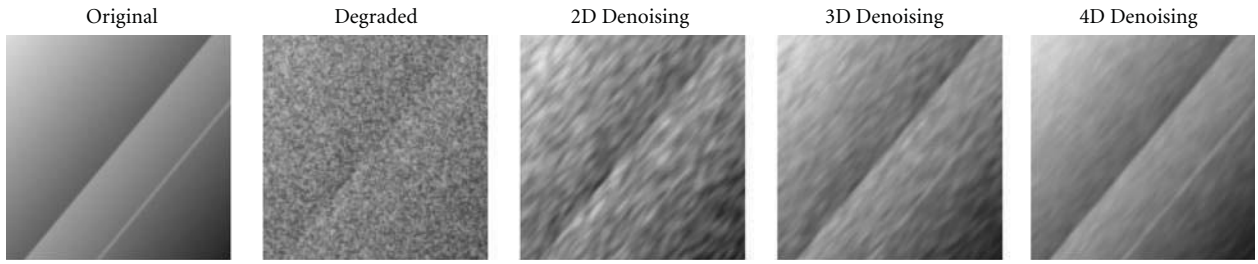


Figure 1: (1) Original test image without noise. There is a large step in the middle, a bright thin line and a shading from the top left corner to the bottom right corner. (2) Original test image with a lot of noise. The step is barely visible, while it is impossible to see the line or the shading. (3) Resulting image after 2D denoising. The step is almost visible and it is possible to see that the top left corner is brighter than the bottom right corner. (4) Resulting image after 3D denoising. Now the step and the shading are clearly visible, but not the line. (5) Resulting image after 4D denoising. Now all parts of the image are clearly visible.

Second, the tuning of the parameters is for our denoising algorithm rather easy to understand and to explore. When a first denoising result has been obtained, it is often obvious how to change the parameters to improve the result. This is not always the case for other methods. Third, the adaptive filtering approach has been proven to be very robust (it is extremely seldom that a strange result is obtained). Adaptive filtering has been used for 2D image denoising in commercial clinical software for over 20 years and a recent 3D study [31] proves its potential, robustness, and clinical acceptance. The nonlocal means algorithm only works if the data contains several neighbourhoods with similar properties.

2.4. Estimating the Local Structure Tensor Using Quadrature Filters. The local structure tensor can, for example, be estimated by using quadrature filters [5, 30]. Quadrature filters Q are zero in one half of the frequency domain (defined by the direction of the filter) and can be expressed as two polar separable functions, one radial function R and one directional function D,

$$Q(\mathbf{u}) = R(\|\mathbf{u}\|)D(\mathbf{u}), \tag{2}$$

where $\mathbf{u}$ is the frequency variable. The radial function is a lognormal function

$$R(\|\mathbf{u}\|) = \exp\left(C\ln^2\left(\frac{\|\mathbf{u}\|}{u_0}\right)\right), \quad C = \frac{-4}{B^2\ln(2)}, \tag{3}$$

where u_0 is the centre frequency of the filter and B is the bandwidth (in octaves). The directional function depends on the angle θ between the filter direction vector $\hat{\mathbf{n}}$ and the normalized frequency coordinate vector $\mathbf{u}$ as $\cos(\theta)^2$,

$$D(\mathbf{u}) = \begin{cases} (\mathbf{u}^T\hat{\mathbf{n}})^2, & \mathbf{u}^T\hat{\mathbf{n}} > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Quadrature filters are Cartesian nonseparable and complex valued in the spatial domain, the real part is even and in 2D acts as a line detector, while the imaginary part is odd and in 2D acts as an edge detector. In 3D, the even and odd filters correspond to a plane detector and a 3D edge detector. In 4D, the plane and 3D edge may in addition be time varying. The complex-valued filter response q is an estimate of a bandpass filtered version of the analytical signal with magnitude A and phase ϕ,

$$q = A(\cos(\phi) + i\cdot\sin(\phi)) = Ae^{i\phi}. \tag{5}$$

The tensor is calculated by multiplying the magnitude of the quadrature filter response q_k with the outer product of the filter direction vector $\hat{\mathbf{n}}_k$ and then summing the result over all filters k,

$$\mathbf{T} = \sum_{k=1}^{N_f} |q_k| \left(c_1\hat{\mathbf{n}}_k\hat{\mathbf{n}}_k^T - c_2\mathbf{I}\right), \tag{6}$$

where c_1 and c_2 are scalar constants that depend on the dimensionality of the data [5, 30], N_f is the number of quadrature filters and $\mathbf{I}$ is the identity matrix. The resulting tensor is *phase invariant*, as the magnitude of the quadrature filter response is invariant to the type of local neighbourhood (e.g., in 2D bright lines, dark lines, dark to bright edges, etc.). This is in contrast to when the local structure tensor is estimated by using gradient operators, such as Sobel filters.

The number of filters that are required to estimate the tensor depends on the dimensionality of the data and is given by the number of independent components of the symmetric local structure tensor. The required number of filters is thus 3 for 2D, 6 for 3D and 10 for 4D. The given tensor formula, however, assumes that the filters are evenly spread. It is possible to spread 6 filters evenly in 3D, but it is not possible to spread 10 filters evenly in 4D. For this reason, 12 quadrature filters have to be used in 4D (i.e., a total of 24 filters in the spatial domain, 12 real valued and 12 complex valued). To apply 24 nonseparable filters to a 4D dataset requires a huge number of multiplications. In this paper a new type of filters, *monomial* filters [32], are therefore used instead.

2.5. Estimating the Local Structure Tensor Using Monomial Filters. Monomial filters also have one radial function R and one directional function D. The directional part of the monomial filters are products of positive integer powers of

the components of the frequency variable $\mathbf{u}$. The monomial filter matrices of order one, $\mathbf{F_1}$, and two, $\mathbf{F_2}$, are in the frequency domain defined as

$$\mathbf{F_{1,n}} = R(\|\mathbf{u}\|)\hat{u}_n, \qquad \mathbf{F_{2,mn}} = R(\|\mathbf{u}\|)\hat{u}_m\hat{u}_n. \tag{7}$$

The monomial filters are first described for 2D and then generalized to 4D.

2.5.1. Monomial Filters in 2D. In 2D, the frequency variable is in this work defined as $\mathbf{u} = [u\ v]^T$. The directional part of first-order monomial filters are x, y in the spatial domain and u, v in the frequency domain. Two-dimensional monomial filters of the first-order are given in Figure 2. The directional part of second-order monomial filters are xx, xy, yy in the spatial domain and uu, uv, vv in the frequency domain. Two dimensional monomial filters of the second order are given in Figure 3.

The monomial filter response matrices $\mathbf{Q}$ are either calculated by convolution in the spatial domain or by multiplication in the frequency domain. For a simple signal with phase θ (e.g., $\mathbf{s}(\mathbf{x}) = A\cos(\mathbf{u}^T\mathbf{x} + \theta)$); the monomial filter response matrices of order one and two can be written as

$$\mathbf{Q_1} = -iA\sin(\theta)[uv]^T,$$
$$\mathbf{Q_2} = A\cos(\theta)\begin{pmatrix} uu & uv \\ uv & vv \end{pmatrix}. \tag{8}$$

The first-order products are odd functions and are thereby related to the odd sine function, the second order products are even functions and are thereby related to the even cosine function (note the resemblance with quadrature filters that have one even real part and one odd imaginary part). By using the fact that $u^2 + v^2 = 1$, the outer products of the filter response matrices give

$$\mathbf{Q_1}\mathbf{Q_1}^T = \sin^2(\theta)|A|^2\begin{pmatrix} uu & uv \\ uv & vv \end{pmatrix},$$
$$\mathbf{Q_2}\mathbf{Q_2}^T = \cos^2(\theta)|A|^2\begin{pmatrix} uu & uv \\ uv & vv \end{pmatrix}. \tag{9}$$

The local structure tensor $\mathbf{T}$ is then calculated as

$$\mathbf{T} = \mathbf{Q_1}\mathbf{Q_1}^T + \mathbf{Q_2}\mathbf{Q_2}^T = |A|^2\begin{pmatrix} uu & uv \\ uv & vv \end{pmatrix}. \tag{10}$$

From this expression, it is clear that the estimated tensor, as previously, is phase invariant as the square of one odd part and the square of one even part are combined. For information about how to calculate the tensor for higher-order monomials, see our recent work [32].

2.5.2. Monomial Filters in 4D. A total of 14 nonseparable 4D monomial filters (4 odd of the first-order (x, y, z, t) and 10 even of the second-order $(xx, xy, xz, xt, yy, yz, yt, zz, zt, tt)$)

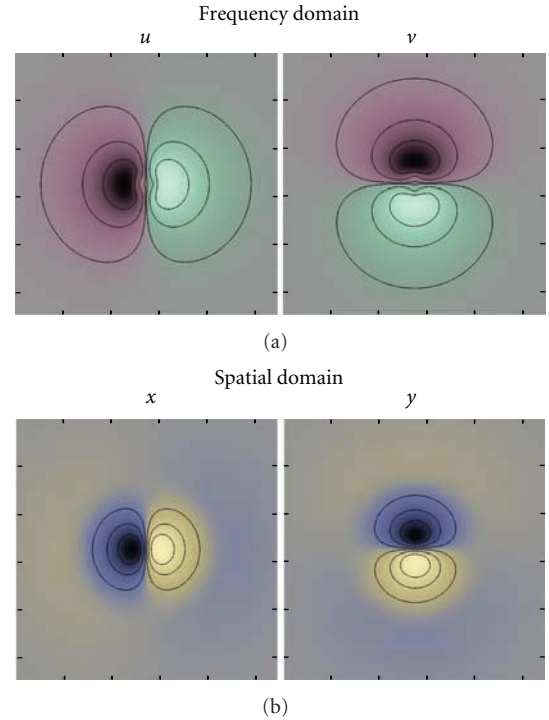


Figure 2: (a) Two-dimensional monomial filters (u, v), of the first order, in the frequency domain. Green indicates positive real values and red indicates negative real values. The black lines are isocurves. (b) Two-dimensional monomial filters (x, y), of the first order, in the spatial domain. Yellow indicates positive imaginary values, and blue indicates negative imaginary values. Note that these filters are odd and imaginary.

with a spatial support of $7 \times 7 \times 7 \times 7$ time voxels are applied to the CT volumes. The filters have a lognormal radial function with centre frequency $3\pi/5$ and a bandwidth of 2.5 octaves. The filter kernels were optimized with respect to ideal frequency response, spatial locality, and expected signal-to-noise ratio [5, 33].

By using equation (10) for the 4D case, and replacing the frequency variables with the monomial filter responses, the 10 components of the structure tensor are calculated according to

$$\begin{aligned}
t_1 &= fr_1 \cdot fr_1 + fr_5 \cdot fr_5 + fr_6 \cdot fr_6 + fr_7 \cdot fr_7 \\
&\quad + fr_8 \cdot fr_8, \\
t_2 &= fr_1 \cdot fr_2 + fr_5 \cdot fr_6 + fr_6 \cdot fr_9 + fr_7 \cdot fr_{10} \\
&\quad + fr_8 \cdot fr_{11}, \\
t_3 &= fr_1 \cdot fr_3 + fr_5 \cdot fr_7 + fr_6 \cdot fr_{10} + fr_7 \cdot fr_{12} \\
&\quad + fr_8 \cdot fr_{13}, \\
t_4 &= fr_1 \cdot fr_4 + fr_5 \cdot fr_8 + fr_6 \cdot fr_{11} + fr_7 \cdot fr_{13} \\
&\quad + fr_8 \cdot fr_{14}, \\
t_5 &= fr_2 \cdot fr_2 + fr_6 \cdot fr_6 + fr_9 \cdot fr_9 + fr_{10} \cdot fr_{10} \\
&\quad + fr_{11} \cdot fr_{11},
\end{aligned}$$

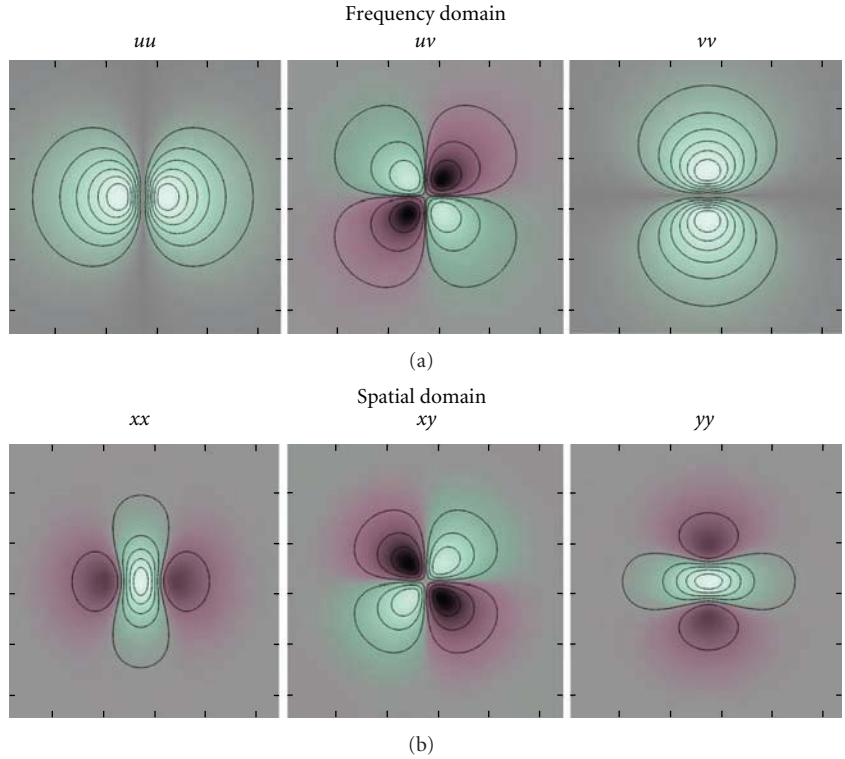


Figure 3: (a) Two-dimensional monomial filters (uu, uv, v), of the second order, in the frequency domain. Green indicates positive real values, and red indicates negative real values. The black lines are isocurves. (b) Two-dimensional monomial filters (xx, xy, yy), of the second order, in the spatial domain. Green indicates positive real values, and red indicates negative real values. Note that these filters are even and real.

$$\begin{aligned}
t_6 &= fr_2 \cdot fr_3 + fr_6 \cdot fr_7 + fr_9 \cdot fr_{10} + fr_{10} \cdot fr_{12} \\
&\quad + fr_{11} \cdot fr_{13}, \\
t_7 &= fr_2 \cdot fr_4 + fr_6 \cdot fr_8 + fr_9 \cdot fr_{11} + fr_{10} \cdot fr_{13} \\
&\quad + fr_{11} \cdot fr_{14}, \\
t_8 &= fr_3 \cdot fr_3 + fr_7 \cdot fr_7 + fr_{10} \cdot fr_{10} + fr_{12} \cdot fr_{12} \\
&\quad + fr_{13} \cdot fr_{13}, \\
t_9 &= fr_3 \cdot fr_4 + fr_7 \cdot fr_8 + fr_{10} \cdot fr_{11} + fr_{12} \cdot fr_{13} \\
&\quad + fr_{13} \cdot fr_{14}, \\
t_{10} &= fr_4 \cdot fr_4 + fr_8 \cdot fr_8 + fr_{11} \cdot fr_{11} + fr_{13} \cdot fr_{13} \\
&\quad + fr_{14} \cdot fr_{14},
\end{aligned} \tag{11}$$

where fr_k denotes the filter response for monomial filter k. The first term relates to $\mathbf{Q_1 Q_1}^T$, and the rest of the terms relate to $\mathbf{Q_2 Q_2}^T$, in total $\mathbf{Q_1 Q_1}^T + \mathbf{Q_2 Q_2}^T$.

If monomial filters are used instead of quadrature filters, the required number of 4D filters is thus decreased from 24 to 14. Another advantage is that the monomial filters require a smaller spatial support, which makes it easier to preserve details and contrast in the processing. A smaller spatial support also results in a lower number of filter coefficients, which decreases the processing time.

2.6. The Control Tensor. When the local structure tensor **T** has been estimated, it is then mapped to a control tensor **C**, by mapping the magnitude (energy) and the isotropy of the tensor. The purpose of this mapping is to further improve the denoising. For 2D and 3D image denoising, this mapping can be done by first calculating the eigenvalues and eigenvectors of the structure tensor in each element of the data. The mapping is first described for 2D and then for 4D.

2.6.1. Mapping the Magnitude of the Tensor in 2D. In the 2D case, the magnitude γ_0 of the tensor is calculated as

$$\gamma_0 = \sqrt{\lambda_1^2 + \lambda_2^2}, \tag{12}$$

where λ_1 and λ_2 are the two eigenvalues. The magnitude γ_0 is normalized to vary between 0 and 1 and is then mapped to γ with a so-called M-function according to

$$\gamma = \left(\frac{\gamma_0^{\beta}}{\gamma_0^{\alpha+\beta} + \sigma^{\beta}} \right), \tag{13}$$

where α, β, and σ are parameters that are used to control the mapping. The σ variable is directly proportional to the signal-to-noise (SNR) ratio of the data and acts as a soft

noise threshold, α mainly controls the overshoot (that can be used for dynamic range compression or to amplify areas that have a magnitude slightly above the noise threshold), and β mainly controls the slope/softness of the curve. The purpose of this mapping is to control the general usage of highpass information. The highpass information should only be used where there is a well-defined structure in the data. If the magnitude of the structure tensor is low, one can assume that the neighbourhood only contains noise. Some examples of the M-function are given in Figure 4.

2.6.2. Mapping the Isotropy of the Tensor in 2D. The isotropy ϕ_0 is in 2D calculated as

$$\phi_0 = \frac{\lambda_2}{\lambda_1} \tag{14}$$

and is mapped to ϕ with a so called mu-function according to

$$\phi = \frac{(\phi_0(1-\alpha))^\beta}{(\phi_0(1-\alpha))^\beta + (\alpha(1-\phi_0))^\beta}, \tag{15}$$

where α and β are parameters that are used to control the mapping, α mainly controls the transition of the curve and β mainly controls the slope/softness. The purpose of this mapping is to control the usage of highpass information in the nondominant direction, that is, the direction that is given by the eigenvector corresponding to the smallest eigenvalue. This is done by making the tensor more isotropic if it is slightly isotropic, or making it even more anisotropic if it is anisotropic. Some examples of the mu-function are given in Figure 5. Some examples of isotropy mappings are given in Figure 6. The M-function and the mu-function are further explained in [5].

2.6.3. The Tensor Mapping in 2D. The control tensor $\mathbf{C}$ is finally calculated as

$$\mathbf{C} = \gamma \mathbf{e_1} \mathbf{e_1}^T + \gamma\phi \mathbf{e_2} \mathbf{e_2}^T, \tag{16}$$

where $\mathbf{e_1}$ is the eigenvector corresponding to the largest eigenvalue λ_1 and $\mathbf{e_2}$ is the eigenvector corresponding to the smallest eigenvalue λ_2. The mapping thus preserves the eigensystem, but changes the eigenvalues and thereby the shape of the tensor.

2.6.4. The Complete Tensor Mapping in 4D. For matrices of size 2×2 and 3×3, there are direct formulas for how to calculate the eigenvalues and eigenvectors, but for 4×4 matrices, there are no such formulas and this complicates the mapping. It would of course be possible to calculate the eigenvalues and eigenvectors by other approaches, such as the power iteration algorithm, but this would be extremely time consuming as the mapping to the control tensor has to be done in each time voxel. The mapping of the local structure tensor to the control tensor is in this work therefore performed in a way that does not explicitly need the calculation of eigenvalues and eigenvectors. The tensor magnitude is first calculated as

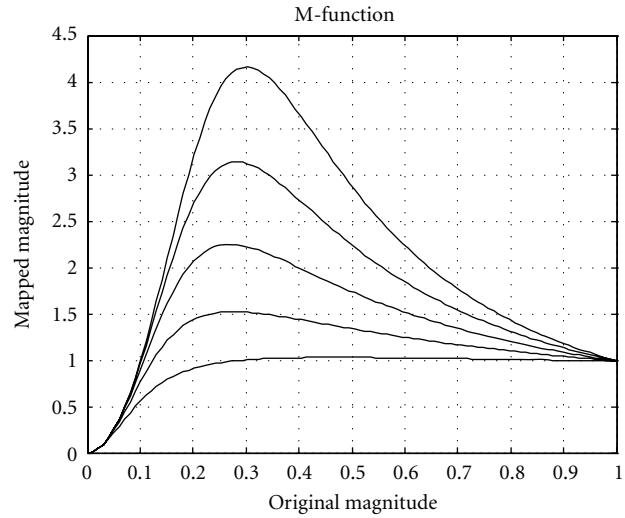


Figure 4: Examples of the M-function that maps the magnitude of the structure tensor. If the magnitude of the structure tensor is too low, the magnitude is set to zero for the control tensor, such that no highpass information is used in this part of the data. The overshoot is intended to amplify structures that have a magnitude that is slightly above the noise threshold.

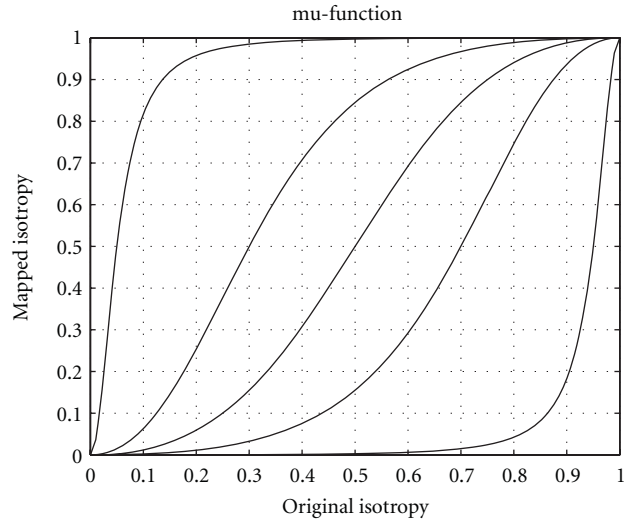


Figure 5: Examples of the mu-function that maps the isotropy of the structure tensor. If the structure tensor is almost isotropic (a high value on the x-axis) the control tensor becomes more isotropic. If the structure tensor is anisotropic (a low value on the x-axis) the control tensor becomes even more anisotropic.

$$\mathbf{T}_{\text{mag}} = ||\mathbf{T}^8||^{1/8}, \tag{17}$$

where $\|\cdot\|$ denotes the Frobenius norm. The exponent will determine how close to λ_1 the estimated tensor magnitude will be; a higher exponent will give better precision, but an

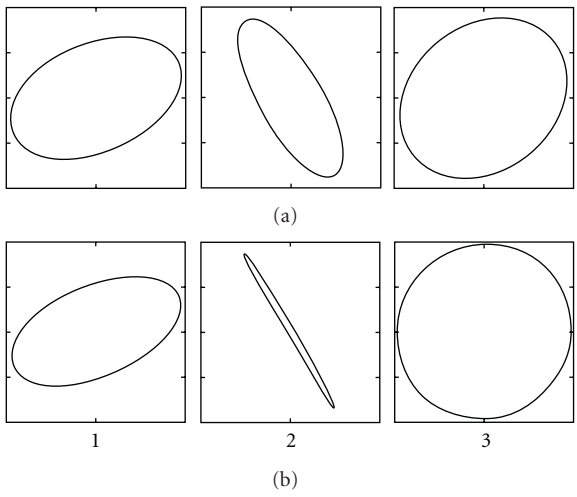


Figure 6: Three examples of isotropy mappings. (a) Original structure tensors. (b) Mapped control tensors. If the structure tensor is anisotropic, the control tensor becomes even more anisotropic (examples 1 and 2). If the structure tensor is almost isotropic, it becomes more isotropic (example 3).

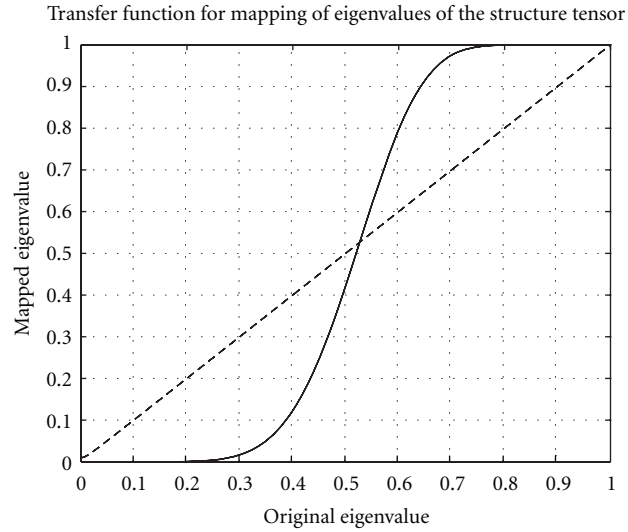


Figure 7: The transfer function that maps the eigenvalues of the structure tensor.

exponent of 8 has proven to be sufficient in practice. To reduce the computational load, $\mathbf{T}^8$ is calculated as

$$\begin{aligned}\mathbf{T}^2 &= \mathbf{T} * \mathbf{T},\\ \mathbf{T}^4 &= \mathbf{T}^2 * \mathbf{T}^2,\\ \mathbf{T}^8 &= \mathbf{T}^4 * \mathbf{T}^4,\end{aligned} \tag{18}$$

where $*$ denotes matrix multiplication. γ_0 is then calculated as

$$\gamma_0 = \sqrt{\frac{\mathbf{T}_{\text{mag}}}{\max\left(\mathbf{T}_{\text{mag}}\right)}}, \tag{19}$$

where the max operator is for the entire data set, such that the maximum value of γ_0 will be 1, γ_0 is then mapped to γ by using the M-function.

To map the isotropy, the structure tensor is first normalized as

$$\hat{\mathbf{T}} = \frac{\mathbf{T}}{\mathbf{T}_{\text{mag}}}, \tag{20}$$

such that the tensor only carries information about the anisotropy (shape). The fact that $\hat{\mathbf{T}}$ and $\mathbf{I} - \hat{\mathbf{T}}$ have the same eigensystem is used, such that the control tensor can be calculated as

$$\mathbf{C} = \gamma\left(\phi\mathbf{I} + (1-\phi)\cdot\hat{\mathbf{T}}\right), \tag{21}$$

where $\mathbf{I}$ is the identity matrix. The following formulas are an ad hoc modification of this basic idea, that do not explicitly need the calculation of the isotropy ϕ and that give good results for our CT data. The basic idea is that the ratio of the eigenvalues of the tensor change when the tensor is multiplied with itself a number of times, and thereby the shape of the tensor also changes. This approach does not give exactly the same results as the original isotropy mapping, but it circumvents the explicit calculation of eigenvalues and eigenvectors. A help variable $\hat{\mathbf{T}}_f$ is first calculated as

$$\hat{\mathbf{T}}_f = \hat{\mathbf{T}}^2 * \left(\mathbf{I} + 2\cdot\left(\mathbf{I} - \hat{\mathbf{T}}\right)\right), \tag{22}$$

and then the control tensor $\mathbf{C}$ is calculated as

$$\mathbf{C} = \gamma\left(\mathbf{I} - \left(\mathbf{I} - \hat{\mathbf{T}}_f\right)^8 * \left(\mathbf{I} + 8\cdot\hat{\mathbf{T}}_f\right)\right). \tag{23}$$

The resulting transfer function that maps each eigenvalue is given in Figure 7. Eigenvalues that are small become even smaller, and eigenvalues that are large become even larger. The result of this eigenvalue mapping is similar to the isotropy mapping examples given in Figure 6.

2.7. Calculating the Denoised Data. Eleven nonseparable reconstruction filters, one lowpass filter $\mathbf{H_0}$ of the zeroth order and 10 highpass filters $\mathbf{H_{2,mn}}$ of the second order, with a spatial support of $11\times11\times11\times11$ time voxels are applied to the CT volumes. The denoised 4D data i_{d} is calculated as the sum of the lowpass-filtered data, i_{lp}, and the highpass filtered data for each highpass-filter k, $i_{\text{hp}(k)}$, weighted with the components $\mathbf{C}_k$ of the control tensor $\mathbf{C}$,

$$i_{\text{d}} = i_{\text{lp}} + \sum_{k=1}^{10}\mathbf{C}_k\cdot i_{\text{hp}(k)}. \tag{24}$$

The result is that the 4D data is lowpass filtered in all directions and then highpass information is put back where there is a well-defined structure. Highpass information is put back in the dominant direction of the local neighbourhood (given by the eigenvector related to the largest eigenvalue) if the tensor magnitude is high. Highpass information is put back in the nondominant directions (given by the eigenvectors related to the smaller eigenvalues) if the tensor magnitude is high and the anisotropy is low.

Table 2: The table shows the in and out data resolution, the used equations and the memory consumption for all the processing steps for spatial filtering (SF) and FFT-based filtering (FFTBF). Note that the driver for the GPU is stored in the global memory, and it normally requires 100–200 MB.

Processing step	Resolution, SF	Memory consumption, SF	Resolution, FFTBF	Memory consumption, FFTBF
Lowpass filtering and downsampling of CT volumes	in $512 \times 512 \times 51 \times 20$ out $256 \times 256 \times 26 \times 20$	406 MB	in $512 \times 512 \times 31 \times 20$ out $256 \times 256 \times 16 \times 20$	294 MB
Filtering with 14 monomial filters and calculating the local structure tensor ((10), (11))	in $256 \times 256 \times 26 \times 20$ out $256 \times 256 \times 20 \times 20$	1376 MB	in $256 \times 256 \times 16 \times 20$ out $256 \times 256 \times 10 \times 20$	1791 MB
Lowpass filtering of the local structure tensor components (normalized convolution, (25))	in $256 \times 256 \times 20 \times 20$ out $256 \times 256 \times 20 \times 20$	1276 MB	in $256 \times 256 \times 10 \times 20$ out $256 \times 256 \times 10 \times 20$	720 MB
Calculating the tensor magnitude and mapping it with the M-function ((17), (18), (19), (13))	in $256 \times 256 \times 20 \times 20$ out $256 \times 256 \times 20 \times 20$	1376 MB	in $256 \times 256 \times 10 \times 20$ out $256 \times 256 \times 10 \times 20$	770 MB
Mapping the local structure tensor to the control tensor ((20), (22), (23))	in $256 \times 256 \times 20 \times 20$ out $256 \times 256 \times 20 \times 20$	1376 MB	in $256 \times 256 \times 10 \times 20$ out $256 \times 256 \times 10 \times 20$	770 MB
Lowpass filtering of the control tensor components (normalized convolution, (25))	in $256 \times 256 \times 20 \times 20$ out $256 \times 256 \times 20 \times 20$	1476 MB	in $256 \times 256 \times 10 \times 20$ out $256 \times 256 \times 10 \times 20$	820 MB
Filtering with 11 reconstruction filters, interpolating the control tensor on the fly, and calculating the denoised data (24)	in $512 \times 512 \times 51 \times 20$ out $512 \times 512 \times 39 \times 20$	2771 MB	in $512 \times 512 \times 16 \times 20$ out $512 \times 512 \times 6 \times 20$ (three rounds×6 slices = 18 denoised slices in total)	2110 MB

2.8. The Complete Algorithm. All the processing steps of the denoising algorithm are given in Table 2. In our case the CT data does not contain any significant structural information in the frequencies over $\pi/2$ in the spatial dimensions, the volumes are therefore lowpass filtered and then downsampled a factor 2 in x, y, z. When the local structure tensor has been estimated, it is lowpass filtered, with a separable lowpass filter of size $5 \times 5 \times 5 \times 3$, to improve the estimate in each time voxel and to make sure that the resulting reconstruction filter varies smoothly. Note that this smoothing does not decrease the resolution of the *image data*, but only the resolution of the *tensor field*. After the tensor mapping, the control tensor is interpolated to the original resolution of the CT data.

While the presented algorithm is straightforward to implement, spatial filtering with 11 reconstruction filters of size $11 \times 11 \times 11 \times 11$ (14 641 filter coefficients) applied to a dataset of the resolution $512 \times 512 \times 445 \times 20$ requires about 375 000 billion multiplications. This is the reason why the GPU is needed in order to do the 4D denoising in a reasonable amount of time.

2.9. Normalized Convolution. One of the main drawbacks of the presented algorithm is that, using standard convolution, the number of valid elements in the z-direction (i.e., slices) decreases rapidly. If the algorithm is applied to a dataset of the resolution $512 \times 512 \times 34 \times 20$, two slices are first lost due to the convolution with the lowpass filter of size $3 \times 3 \times 3$. After the downsampling, there are 16 slices in the data. The monomial filters are of size $7 \times 7 \times 7 \times 7$, thereby only 10 of the filter response slices are valid. During the lowpass filtering of each structure tensor component, another four slices are lost and then another four are lost during lowpass filtering of the control tensor. The result is thus that only 2 valid slices are left after all the convolutions. The same problem could exist in the time dimension, but since the heart cycle is periodic it is natural to use circular convolution in the time direction, and thereby all the time points are valid.

The loss of valid slices can be avoided by using normalized convolution [34], both for the lowpass filtering of the data before downsampling and the lowpass filtering of the tensor components. In normalized convolution, a certainty is attached to each signal value. A certainty-weighted filter response cwr is calculated as

$$\text{cwr} = \frac{(c \cdot s) * f}{c * f}, \tag{25}$$

where c is the certainty, s is the signal, f is the filter, $\cdot$ denotes pointwise multiplication, and $*$ denotes convolution. The certainty is set to 1 inside the data and 0 outside the data. Note that this simple version of normalized convolution (normalized averaging) can not be applied for the monomial filters and for the reconstruction filters, as these filters have both negative and positive coefficients. It is possible to apply the full normalized convolution approach for these filters, but it will significantly increase the computational load.

3. GPU Implementation

In this section, the GPU implementation of the denoising algorithm will be described. The CUDA (compute unified device architecture) programming language by Nvidia [35],

explained by Kirk and Hwu [36], has been used for the implementation. The Open Computing Language (OpenCL) [37] could be a better choice, as it makes it possible to run the same code on any hardware.

3.1. Creating 4D Indices. The CUDA programming language can easily generate 2D indices for each thread, for example, by using Algorithm 1. To generate 3D indices is harder, as each thread block can be *three* dimensional but the grid can only be *two* dimensional. One approach to generate 3D indices is given in Algorithm 2. To generate 4D indices is even more difficult. To navigate in the 4D data, the 3D indexing approach described above is used, and the kernel is then called once for each time point.

3.2. Spatial versus FFT Based Filtering. Fast-Fourier-transform (FFT-) based filtering can be very efficient when large nonseparable filters of high dimension are to be applied to big datasets, but spatial filtering is generally faster if the filters are small or Cartesian separable. The main advantage with FFT-based filtering is that the processing time is the same regardless of the spatial size of the filter. A small bonus is that circular filtering is achieved for free. The main disadvantage with FFT-based filtering is however the memory requirements, as the filters need to be stored in the same resolution as the data, and also as a complex-valued number for each element.

To see which kind of filtering that fits the GPU best, both spatial and FFT-based filtering was therefore implemented. For filtering with the small separable lowpass filters (which are applied before the data is downsampled and to smooth the tensor components), only separable spatial filtering is implemented.

3.3. Spatial Filtering. Spatial filtering can be implemented in rather many ways, especially in four dimensions. One easy way to implement 2D and 3D filtering on the GPU is to take advantage of the cache of the texture memory and put the filter kernel in constant memory. The drawback with this approach is however that the implementation will be very limited by the memory bandwidth, and not by the computational performance. Another problem is that it is not possible to use 4D textures in the CUDA programming language. One would have to store the 4D data as one big 1D texture or as several 2D or 3D textures. A better approach is to take advantage of the shared memory, which increased a factor 3 in size between the Nvidia GTX 285 and the Nvidia GTX 580. The data is first read into the shared memory and then the filter responses are calculated in parallel. By using the shared memory, the threads can share the data in a very efficient way, which is beneficial as the filtering results for two neighbouring elements are calculated by mainly using the same data.

As multidimensional filters can be separable or nonseparable (the monomial filters and the reconstruction filters are nonseparable, while the different lowpass filters are separable) two different spatial filtering functions were implemented.

3.3.1. Separable Filtering. Our separable 4D convolver is implemented by first doing the filtering for all the rows, then for all the columns, then for all the rods and finally for all the time points. The data is first loaded into the shared memory and then the valid filter responses are calculated in parallel. The filter kernels are stored in constant memory. For the four kernels, 16 KB of shared memory is used such that 3 thread blocks can run in parallel on each multiprocessor on the Nvidia GTX 580.

3.3.2. Nonseparable Filtering. The shared memory approach works rather well for nonseparable 2D filtering but not as well for nonseparable 3D and 4D filtering. The size of the shared memory on the Nvidia GTX 580 is 48 KB for each multiprocessor, and it is thereby only possible to, for example, fit $11 \times 11 \times 11 \times 9$ float values into it. If the 4D filter is of size $9 \times 9 \times 9 \times 9$, only $3 \times 3 \times 3 \times 1 = 27$ valid filter responses can be generated for each multiprocessor. A better approach for nonseparable filtering in 4D is to instead use an optimized 2D filtering kernel, and then accumulate the filter responses by summing over the other dimensions by calling the 2D filtering function for each slice and each time point of the filter. The approach is described with the pseudocode given in Algorithm 3.

Our nonseparable 2D convolver first reads 64×64 pixels into the shared memory, then calculates the valid filter responses for all the 14 monomial filters or all the 11 reconstruction filters at the same time, and finally writes the results to global memory. Two versions of the convolver were implemented, one that maximally supports 7×7 filters and one that maximally supports 11×11 filters. The first calculates 58×58 valid filter responses, and the second calculates 54×54 valid filter responses. As 64×64 float values only require 16 KB of memory, three thread blocks can run at the same time on each multiprocessor. This results in $58 \times 58 \times 3 = 10092$ and $54 \times 54 \times 3 = 8748$ valid filter responses per multiprocessor. For optimal performance, the 2D filtering loop was completely unrolled by generating the code with a Matlab script.

The 14 monomial filters are of size $7 \times 7 \times 7 \times 7$, this would require 135 KB of memory to be stored as floats, but the constant memory is only 64 KB. For this reason, 7×7 filter coefficients are stored at a time and are then updated for each time point and for each slice. It would be possible to store $7 \times 7 \times 7$ filter coefficients at a time, but by only storing 7×7 coefficients, the size of the filters (2.75 KB) is small enough to always be in the cache of the constant memory (8 KB). The same approach is used for the 11 reconstruction filters of size $11 \times 11 \times 11 \times 11$.

3.4. FFT-Based Filtering. While the CUFFT library by Nvidia supports 1D, 2D, and 3D FFTs, there is no direct support for 4D FFTs. As the FFT is cartesian separable, it is however possible to do a 4D FFT by applying four consecutive 1D FFTs. The CUFFT library supports launching a batch of 1D FFTs, such that many 1D FFT's can run in parallel. The batch of 1D FFTs are applied along the first dimension in which the data is stored (e.g., along x if the data is stored

```
// Code that is executed before the kernel is launched
int threadsInX = 32;
int threadsInY = 16;

int blocksInX = DATA_W/threadsInX;
int blocksInX = DATA_H/threadsInY;

dimGrid = dim3(blocksInX, blocksInY);
dimBlock = dim3(threadsInX, threadsInY, 1);

// Code that is executed inside the kernel
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIx.yd;
```

Algorithm 1

```
// Code that is executed before the kernel is launched
int threadsInX = 32;
int threadsInY = 16;
int threadsInZ = 1;

int blocksInX = (DATA_W+threadsInX-1)/threadsInX;
int blocksInY = (DATA_H+threadsInY-1)/threadsInY;
int blocksInZ = (DATA_D+threadsInZ-1)/threadsInZ;
dim3 dimGrid = dim3(blocksInX, blocksInY*blocksInZ);
dim3 dimBlock = dim3(threadsInX, threadsInY, threadsInZ);

// Code that is executed inside the kernel
int blockIdxz = __float2uint_rd(blockIdx.y * invBlocksInY);
int blockIdxy = blockIdx.y - blockIdxz * blocksInY;
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdxy * blockDim.y + threadIdx.y;
int z = blockIdxz * blockDim.z + threadIdx.z;
```

Algorithm 2

as (x, y, z, t)). Between each 1D FFT, it is thereby necessary to change the order of the data (e.g., from (x, y, z, t) to (y, z, t, x)). The drawback with this approach is that the time it takes to change order of the data can be longer than to actually perform the 1D FFT. The most recent version of the CUFFT library supports launching a batch of 2D FFT's. By applying two consecutive 2D FFT's, it is sufficient to change the order of the data once, instead of three times.

A forward 4D FFT is first applied to the volumes. A filter is padded with zeros to the same resolution as the data and is then transformed to the frequency domain. To do the filtering, a complex-valued multiplication between the data and the filter is applied and then an inverse 4D FFT is applied to the filter response. After the inverse transform, a FFT shift is necessary; there is however no such functionality in the CUFFT library. When the tensor components and the denoised data are calculated, each of the four coordinates is shifted by using a help function, see Algorithm 4.

As the monomial filters only have a real part or an imaginary part in the spatial domain, some additional time is saved by putting one monomial filter in the real part and another monomial filter in the imaginary part before the 4D FFT is applied to the zero-padded filter. When the complex multiplication is performed in the frequency domain, two filters are thus applied at the same time. After the inverse 4D FFT, the first filter response is extracted as the real part and second filter response is extracted as the imaginary part. The same trick is used for the 10 highpass reconstruction filters.

3.5. Memory Considerations. The main problem of implementing the 4D denoising algorithm on the GPU is the limited size of the global memory (3 GB in our case). This is made even more difficult by the fact that the GPU driver can use as much as 100–200 MB of the global memory. Storing all the CT data on the GPU at the same time is not possible, a single CT volume of the resolution $512 \times 512 \times 445$ requires about 467 MB of memory if 32 bit floats are used. Storing the filter responses is even more problematic. To give an example, to store all the 11 reconstruction filter responses as floats for a dataset of the size $512 \times 512 \times 445 \times 20$ would require about 103 GB of memory. The denoising is therefore done for a number of slices (e.g., 16 or 32) at a time.

For the spatial filtering, the algorithm is started with data of the resolution $512 \times 512 \times 51 \times 20$ and is downsampled to $256 \times 256 \times 26 \times 20$. The control tensor is calculated for $256 \times 256 \times 20 \times 20$ time voxels, and the denoised data is calculated for $512 \times 512 \times 39 \times 20$ time voxels. To process all the 445 slices requires 12 runs.

```
// Do the filtering for all the time points in the data
for (int t=0; t<DATA_T; t++)
{
    // Do the filtering for all the slices in the data
    for (int z=0; z<DATA_D;z++)
    {
        // Set the filter responses on the GPU to 0
        Reset<<<dimGrid, dimBlock>>>(d_Filter_Responses);
        // Do the filtering for all the time points in the filter
        for (int tt=0; tt<FILTER_T; tt++)
        {
            // Do the filtering for all the slices in the filter
            for (int zz=0; zz<FILTER_D; zz++)
            {
                // Copy the current filter coefficients
                // to constant memory on the GPU
                CopyFilterCoefficients(zz,tt);
                // Do the 2D filtering on the GPU
                // and increment the filter responses
                // inside the filtering function
                Conv2D<<<dimGrid, dimBlock>>>(d_Filter_Responses);
            }
        }
    }
}
```

Algorithm 3

```
__device__ int Shift_FFT_Coordinate(int coordinate, int DATA_SIZE)
{
   if (coordinate > (ceilf(DATA_SIZE/2) - 1))
   {
     return coordinate - ceilf(DATA_SIZE/2);
   }
   else
   {
     return coordinate + floorf(DATA_SIZE/2);
   }
}
```

Algorithm 4

For the FFT-based filtering, the algorithm is started with data of the resolution $512\times512\times31\times20$ and is downsampled to $256 \times 256 \times 16 \times 20$. The control tensor is then calculated for $256 \times 256 \times 10 \times 20$ time voxels, and the denoised data is calculated for $512 \times 512 \times 18 \times 20$ time voxels. To process all the 445 slices requires 26 runs.

To store the 10 components of the control tensor in the same resolution as the original CT data for one run with spatial filtering ($512 \times 512 \times 39 \times 20$) would require about 12.2 GB of memory. As the control tensor needs to be interpolated a factor 2 in each spatial dimension, since it is estimated on downsampled data, another approach is used. Interpolating the tensor is a perfect task for the GPU, due to the hardware support for linear interpolation. The 10 tensor components, for one timepoint, are therefore stored in 10 textures and then the interpolation is done on the fly when the denoised data is calculated. By using this approach, only another 10 variables of the resolution $256 \times 256 \times 20$ need to be stored at the same time.

Table 2 states the in and out resolution of the data, the used equations, and the memory consumption at each step of the denoising algorithm, for spatial filtering and FFT-based filtering. The out resolution refers to the resolution of the data that is valid after each processing step, as some data is regarded as non-valid after filtering operations. The reason why the memory consumption is larger for the FFT-based

Table 3: Processing times for filtering with the 14 monomial filters of size $7 \times 7 \times 7 \times 7$ and calculating the 4D tensor for the different implementations. The processing times for the GPU do not include the time it takes to transfer the data to and from the GPU.

Data size	Spatial filtering CPU	Spatial filtering GPU	GPU speedup	FFT filtering CPU	FFT filtering GPU	GPU speedup
$128 \times 128 \times 111 \times 20$	17.3 min	5.7 s	182	25 s	1.8 s	13.9
$256 \times 256 \times 223 \times 20$	2.3 h	36.0 s	230	3.3 min	14.3 s	13.9

Table 4: Processing times for lowpass filtering the 10 tensor components, calculating γ and mapping the structure tensor to the control tensor for the different implementations. The processing times for the GPU do not include the time it takes to transfer the data to and from the GPU.

Data size	CPU	GPU	GPU speedup
$256 \times 256 \times 223 \times 20$	42 s	1.0 s	42
$512 \times 512 \times 445 \times 20$	292 s	7.3 s	40

filtering is that the spatial filtering can be done for one slice or one volume at a time, while the FFT-based filtering has to be applied to a sufficiently large number of slices and time points at the same time. We were not able to use more than about 2 GB of memory for the FFT-based filtering; one reason for this might be that the CUFFT functions internally use temporary variables that use some of the memory. Since the source code for the CUFFT library is unavailable, it is hard to further investigate this hypothesis.

4. Data

The 4D CT dataset that was used for testing our GPU implementation was collected with a Siemens SOMATOM Definition Flash dual-energy CT scanner at the Center for medical Image Science and Visualization (CMIV). The dataset contains almost 9000 DICOM files and the resolution of the data is $512 \times 512 \times 445 \times 20$ time voxels. The spatial size of each voxel is $0.75 \times 0.75 \times 0.75$ mm. During the image acquisition the tube current is modulated over the cardiac cycle with reduced radiation exposure during the systolic heart phase. Due to this, the amount of noise varies with time.

5. Results

5.1. Processing Times. A comparison between the processing times for our GPU implementation and for a CPU implementation was made. The used GPU was a Nvidia GTX 580, equipped with 512 processor cores and 3 GB of memory (the Nvidia GTX 580 is normally equipped with 1.5 GB of memory). The used CPU was an Intel Xeon 2.4 GHz with 4 processor cores and 12 MB of L3 cache, 12 GB of memory was used. All the implementations used 32 bit floats. The operating system used was Linux Fedora 14 64-bit.

For the CPU implementation, the OpenMP (open multiprocessing) library [38, 39] was used, such that all the 4 processor cores work in parallel. No other types of optimization for the CPU, such as SSE2, were used. We are fully aware of the fact that it is possible to make a much better CPU implementation. The purpose of this comparison is rather to give an *indication* of the performance of the CPU and the GPU. If the CPU code would be vectorized, the CPU processing times can be divided by a factor 3 or 4 (except for the FFT which already is very optimized).

The processing times are given in Tables 3, 4, 5, and 6. The CPU processing times for the spatial filtering are *estimates*, since it takes several days to run the algorithm on the whole dataset. The processing times for a multi-GPU implementation would scale rather linearly with the number of GPUs, since each GPU can work on different subsets of slices in parallel. As our computer contains three GPUs, all the processing times for the GPU can thereby be divided by a factor 3.

5.2. Denoising Results. To show the results of the 4D denoising, the original CT data was compared with the denoised data by applying volume rendering. The freely available MeVisLab software development program (http://www.mevislab.de/) was used. Two volume renderers, one for the original data and one for the denoised data, run at the same time and were synced in terms of view angle and transfer function. Figure 8 shows volume renderings of the original and the denoised data for different time points and view angels. It is clear that a lot of noise is removed by the denoising, but since the denoising algorithm alters the histogram of the data, it is hard to make an objective comparison even if the same transfer function is applied.

A movie where the original and the denoised data is explored with the two volume renderers was also made. For this video, the data was downsampled a factor 2 in the spatial dimensions, in order to decrease the memory usage. The volume renderers automatically loop over all the time-points. The video can be found at http://www.youtube.com/watch?v=wflbt2sV34M.

By looking at the video, it is easy to see that the amount of noise in the original data varies with time.

6. Discussion

We have presented how to implement true 4D image denoising on the GPU. The result is that 4D image denoising becomes practically possible if the GPU is used and thereby the clinical value increases significantly.

6.1. Processing Times. To make a completely fair comparison between the CPU and the GPU is rather difficult. It has been debated [40] if the GPU speedups that have been reported in the literature are plausible or if they are the result of comparisons with unoptimized CPU implementations. In our opinion, the theoretical and practical processing performance that can be achieved for different hardware is

Table 5: Processing times for filtering with the 11 reconstruction filters of size 11×11×11×11 and calculating the denoised data for the different implementations. The processing times for the GPU do NOT include the time it takes to transfer the data to and from the GPU.

Data size	Spatial filtering CPU	Spatial filtering GPU	GPU speedup	FFT filtering CPU	FFT filtering GPU	GPU speedup
256 × 256 × 223 × 20	7.5 h	3.3 m	136	5.6 min	1.1 min	5.1
512 × 512 × 445 × 20	2.5 days	23.9 m	150	45 min	8.6 min	5.2

Table 6: Total processing times for the complete 4D image denoising algorithm for the different implementations. The processing times for the GPU DO include the time it takes to transfer the data to and from the GPU.

Data size	Spatial filtering CPU	Spatial filtering GPU	GPU speedup	FFT filtering CPU	FFT filtering GPU	GPU speedup
256 × 256 × 223 × 20	7.8 h	3.5 m	133	6.7 m	1.2 m	5.6
512 × 512 × 445 × 20	2.6 days	26.3 m	144	52.8 m	8.9 m	5.9

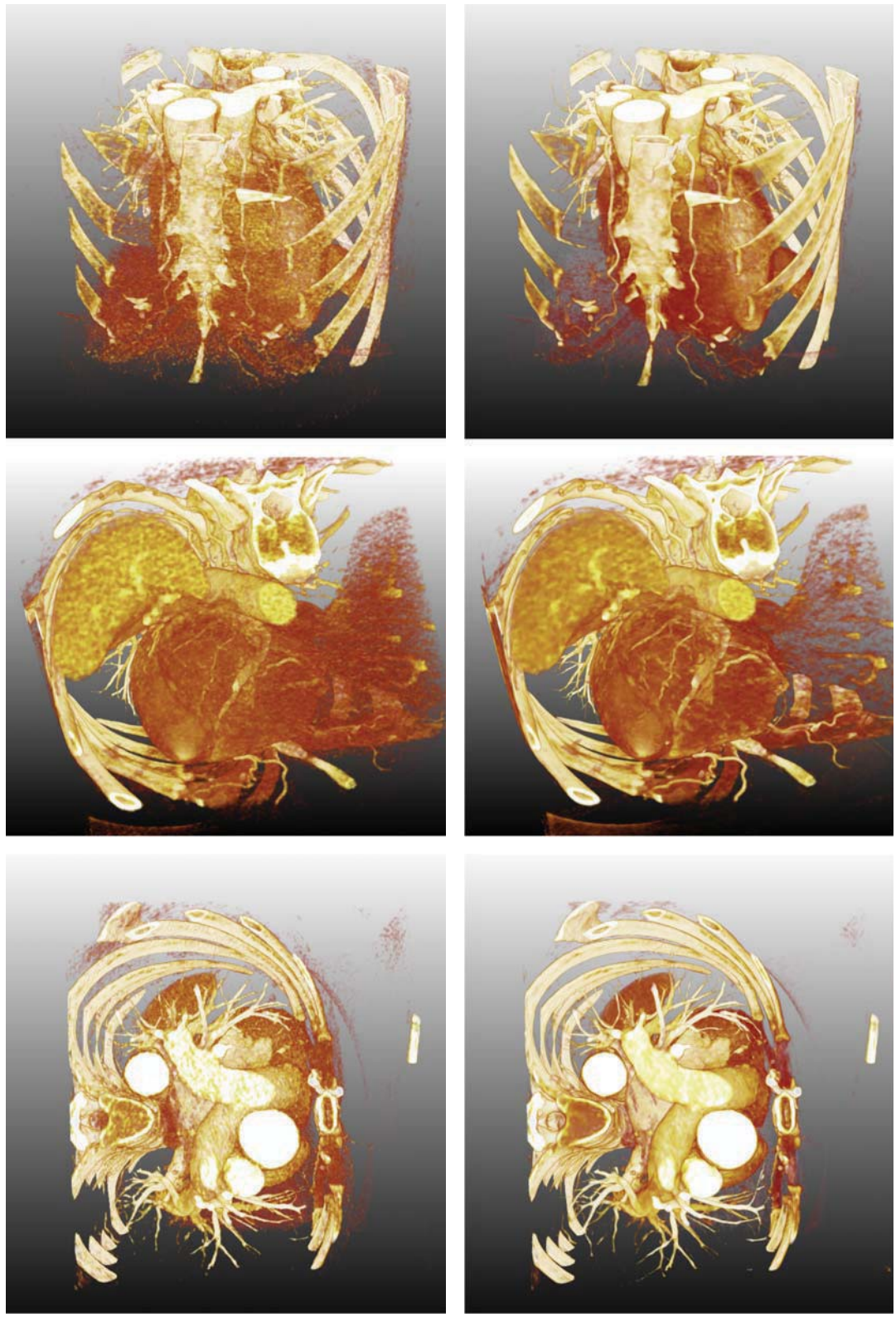
not the only interesting topic. In a research environment, the *ratio* between the achievable processing performance and the time it takes to do the implementation is also important. From this perspective, we think that our CPU-GPU comparison is rather fair, since about the same time was spent on doing the CPU and the GPU implementation. The CUDA programming language was designed and developed for parallel calculations from the beginning, while different addons have been added to the C programming language to be able to do parallel calculations. While it is rather easy to make the CPU implementation multithreaded, for example, by using the OpenMP library, more advanced CPU optimization is often more difficult to include and often requires assembler programming.

While spatial filtering can be significantly slower than FFT-based filtering for nonseparable filters, there are some advantages (except for the lower memory usage). One is that a region of interest (ROI) can be selected for the denoising, compared to doing the denoising on the whole dataset. Another advantage is that filter networks [41, 42] can be applied, such that the filter responses from many small filters are combined to the same filter response as from one large filter. Filter networks can reduce the number of multiplications as much as a factor 5 in 2D, 25 in 3D and 300 in 4D [43]. To design and optimize a filter network however requires much more work than to optimize a single filter [33]. Another problem is that the memory usage increases significantly when filter networks are used, since many filter responses need to be stored in memory. Filter networks on the GPU is a promising area for future research.

From our results, it is clear that FFT-based filtering is faster than spatial filtering for large nonseparable filters. For data sizes that are not a power of two in each dimension, the FFT based approach might however not be as efficient. Since medical doctors normally do not look at 3D or 4D data as volume renderings, but rather as 2D slices, the spatial filtering approach however has the advantage that the denoising can be done for a region of interest (e.g., a specific slice or volume). It is a waste of time to enhance the parts of the data that are not used by the medical doctor. The spatial filtering approach can also handle larger datasets than the FFT-based approach, as it is sufficient to store the filter responses for one slice or one volume at a time. Recently, we acquired a CT data set with 100 time points, compared to 20 time points. It is not possible to use the FFT-based approach for this data set.

There are several reasons why the GPU speedup for the FFT-based filtering is much smaller than the GPU speedup for the spatial filtering. First, the CUFFT library does not include any direct support for 4D FFT's, and we had to implement our own 4D FFT as two 2D FFT's that are applied after each other. Between the 2D FFT's the storage order of the data is changed. It can take a longer time to change the order of the data than to actually perform the FFT. If Nvidia includes direct support for 4D FFT's in the CUFFT library, we are sure that their implementation would be much more efficient than ours. Second, the FFT for the CPU is extremely optimized, as it is used in a lot of applications, and our convolver for the CPU is not fully optimized. The CUDA programming language is only a few years old, and the GPU standard libraries are not as optimized as the CPU standard libraries. The hardware design of the GPUs also changes rapidly. Some work has been done in order to further optimize the CUFFT library. Nukada et al. [44, 45] have created their own GPU FFT library which has been proven to give better performance than the CUFFT library. They circumvent the problem of changing the order of the data and thereby achieve an implementation that is much more efficient. In 2008, their 3D FFT was 5-6 times faster than the 3D FFT in the CUFFT library. Third, due to the larger memory requirements of FFT-based filtering it is not possible to achieve an as big speedup for the GPU implementation as for the CPU implementation. If a GPU with a higher amount of global memory would have been used, the FFT-based implementation would have been more efficient.

6.2. 4D Image Processing with CUDA. As previously discussed in the paper, 4D image processing in CUDA is harder to implement than 2D and 3D image processing. There are, for example, no 4D textures, no 4D FFTs, and there is no direct support for 4D (or 3D) indices. However, since fMRI data also is 4D, we have previously gained some experience on how to do 4D image processing with CUDA [18–20]. The conclusions that we draw after implementing the 4D image denoising algorithm with the CUDA programming language is thus that CUDA is not perfectly suited for 4D image processing, but due to its flexibility, it was still possible to implement the algorithm rather easily.



(a) (b)

Figure 8: Three comparisons between original CT data (a) and denoised CT data (b). The parameters used for this denoising where $\alpha = 0.55$, $\beta = 1.5$, and $\sigma = 0.1$ for the M-function.

6.3. True 5D Image Denoising. It might seem impossible to have medical image data with more than 4 dimensions, but some work has been done on how to collect 5D data [46]. The five dimensions are the three spatial dimensions and *two* time dimensions, one for the breathing rhythm and one for the heart rhythm. One major advantage with 5D data is that the patient can breathe normally during the data acquisition, while the patient has to hold its breath during collection of 4D data. With 5D data, it is possible to, for example, fixate the heart and only see the lungs moving, or fixate the lungs to only see the heart beating. If the presented algorithm would be extended to 5D, it would be necessary to use a total of 20 monomial filters and 16 reconstruction filters. For a 5D dataset of the size $512 \times 512 \times 445 \times 20 \times 20$, the required number of multiplications for spatial filtering with the reconstruction filters would increase from 375 000 billion for 4D to about 119 million billion ($1.19 \cdot 10^{17}$) for 5D. The size of the reconstruction filter responses would increase from 103 GB for 4D to 2986 GB for 5D. This is still only *one* dataset for *one* patient, and we expect that both the spatial and the temporal resolution of all medical imaging modalities will increase even further in the future. Except for the 5 *outer* dimensions, it is also possible to collect data with more than one *inner* dimension. This is, for example, the case if the blood flow of the heart is to be studied. For flow data, a three-dimensional vector needs to be stored in each time voxel, instead of a single intensity value.

7. Conclusions

To conclude, by using the GPU, true 4D image denoising becomes practically feasible. Our implementation can of course be applied to other modalities as well, such as ultrasound and MRI, and not only to CT data. The short processing time also makes it practically possible to further improve the denoising algorithm and to tune the parameters that are used.

The elapsed time between the development of practically feasible 2D [2] and 3D [4] image denoising techniques was about 10 years, from 3D to 4D the elapsed time was about 20 years. Due to the rapid development of GPUs, it is hopefully *not* necessary to wait another 10–20 years for 5D image denoising.

Acknowledgments

This work was supported by the Linnaeus Center CADICS and research Grant no. 2008-3813, funded by the Swedish research council. The CT data was collected at the Center for Medical Image Science and Visualization (CMIV). The authors would like to thank the NovaMedTech project at Linköping University for financial support of the GPU hardware, Johan Wiklund for support with the CUDA installations, and Chunliang Wang for setting the transfer functions for the volume rendering.

References

[1] J.-S. Lee, "Digital image enhancement and noise filtering by use of local statistics," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 2, no. 2, pp. 165–168, 1980.

[2] H. E. Knutsson, R. Wilson, and G. H. Granlund, "Anisotropic non-stationary image estimation and its applications—part I: restoration of noisy images," *IEEE Transactions on Communications*, vol. 31, no. 3, pp. 388–397, 1983.

[3] P. Perona and J. Malik, "Scale-space and edge detection using anisotropic diffusion," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 12, no. 7, pp. 629–639, 1990.

[4] H. Knutsson, L. Haglund, H. Bårman, and G. Granlund, "A framework for anisotropic adaptive filtering and analysis of image sequences and volumes," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP)*, pp. 469–472, 1992.

[5] G. Granlund and H. Knutsson, *Signal Processing for Computer Vision*, Kluwer Academic, Boston, Mass, USA, 1995.

[6] C.-F. Westin, L. Wigström, T. Loock, L. Sjöqvist, R. Kikinis, and H. Knutsson, "Three-dimensional adaptive filtering in magnetic resonance angiography," *Journal of Magnetic Resonance Imaging*, vol. 14, pp. 63–71, 2001.

[7] J. Montagnat, M. Sermesant, H. Delingette, G. Malandain, and N. Ayache, "Anisotropic filtering for model-based segmentation of 4D cylindrical echocardiographic images," *Pattern Recognition Letters*, vol. 24, no. 4-5, pp. 815–825, 2003.

[8] H. Jahanian, A. Yazdan-Shahmorad, and H. Soltanian-Zadeh, "4D wavelet noise suppression of MR diffusion tensor data," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP)*, pp. 509–512, April 2008.

[9] K. Pauwels and M. M. Van Hulle, "Realtime phase-based optical flow on the GPU," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, (CVPR)*, pp. 1–8, June 2008.

[10] P. Muyan-Özcelik, J. D. Owens, J. Xia, and S. S. Samant, "Fast deformable registration on the GPU: a CUDA implementation of demons," in *Proceedings of the International Conference on Computational Sciences and its Applications, (ICCSA)*, pp. 223–233, July 2008.

[11] P. Bui and J. Brockman, "Performance analysis of accelerated image registration using GPGPU," in *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units, (GPGPU-2)*, pp. 38–45, March 2009.

[12] A. Eklund, M. Andersson, and H. Knutsson, "Phase based volume registration using CUDA," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, (ICASSP)*, pp. 658–661, March 2010.

[13] R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley, "A survey of medical image registration on multicore and the GPU," *IEEE Signal Processing Magazine*, vol. 27, no. 2, Article ID 5438962, pp. 50–60, 2010.

[14] A. E. Lefohn, J. E. Cates, and R. T. Whitaker, "Interactive, GPU-based level sets for 3D segmentation," *Lecture Notes in Computer Science*, vol. 2878, pp. 564–572, 2003.

[15] V. Vineet and P. J. Narayanan, "CUDA cuts: fast graph cuts on the GPU," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, (CVPR)*, pp. 1–8, June 2008.

[16] A. Abramov, T. Kulvicius, F. Wörgötter, and B. Dellen, "Real-time image segmentation on a GPU," in *Proceedings of Facing the Multicore-Challenge*, vol. 6310 of *Lecture Notes in Computer Science*, pp. 131–142, Springer, 2011.

[17] D. Gembris, M. Neeb, M. Gipp, A. Kugel, and R. Männer, "Correlation analysis on GPU systems using NVIDIA's CUDA," *Journal of Real-Time Image Processing*, pp. 1–6, 2010.

[18] A. Eklund, O. Friman, M. Andersson, and H. Knutsson, "A GPU accelerated interactive interface for exploratory functional connectivity analysis of fMRI data," in *Proceedings of the IEEE International Conference on Image Processing, (ICIP)*, pp. 1621–1624, 2011.

[19] A. Eklund, M. Andersson, and H. Knutsson, "fMRI analysis on the GPU—possibilities and challenges," *Computer Methods and Programs in Biomedicine*. In press.

[20] A. Eklund, M. Andersson, and H. Knutsson, "Fast random permutation tests enable objective evaluation of methods for single subject fMRI analysis," *International Journal of Biomedical Imaging*, vol. 2011, Article ID 627947, 2011.

[21] M. Rumpf and R. Strzodka, "Nonlinear diffusion in graphics hardware," in *Proceedings of the EG/IEEE TCVG Symposium on Visualization*, pp. 75–84, 2001.

[22] M. Howison, "Comparing GPU implementations of bilateral and anisotropic diffusion filters for 3D biomedical datasets," Tech. Rep. LBNL-3425E, Lawrence Berkeley National Laboratory, Berkeley, Calif, USA.

[23] Y. Su and Z. Xu, "Parallel implementation of wavelet-based image denoising on programmable PC-grade graphics hardware," *Signal Processing*, vol. 90, no. 8, pp. 2396–2411, 2010.

[24] Q. Zhang, R. Eagleson, and T. M. Peters, "GPU-based image manipulation and enhancement techniques for dynamic volumetric medical image visualization," in *Proceedings of the 4th IEEE International Symposium on Biomedical Imaging: From Nano to Macro, (ISBI)*, pp. 1168–1171, April 2007.

[25] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid, ACM transactions on graphics," in *Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference*, no. 103, p. 9, 2007.

[26] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Proceedings of the IEEE 6th International Conference on Computer Vision*, pp. 839–846, January 1998.

[27] F. Fontes, G. Barroso, P. Coupe, and P. Hellier, "Real time ultrasound image denoising," *Journal of Real-Time Image Processing*, vol. 6, pp. 15–22, 2010.

[28] B. Goossens, H. Luong, J. Aelterman, A. Pizurica, and W. Philips, "A GPU-accelerated real-time NLMeans algorithm for denoising color video sequences," in *Proceedings of the 12th International Conference on Advanced Concepts for Intelligent Vision Systems, (ACIVS)*, vol. 6475 of *Lecture Notes in Computer Science*, pp. 46–57, Springer, 2010.

[29] A. Buades, B. Coll, and J. M. Morel, "A non-local algorithm for image denoising," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, (CVPR)*, pp. 60–65, June 2005.

[30] H. Knutsson, "Representing local structure using tensors," in *Proceedings of the Scandinavian Conference on Image Analysis, (SCIA)*, pp. 244–251, 1989.

[31] F. Forsberg, V. Berghella, D. A. Merton, K. Rychlak, J. Meiers, and B. B. Goldberg, "Comparing image processing techniques for improved 3-dimensional ultrasound imaging," *Journal of Ultrasound in Medicine*, vol. 29, no. 4, pp. 615–619, 2010.

[32] H. Knutsson, C.-F. Westin, and M. Andersson, "Representing local structure using tensors II," in *Proceedings of the Scandinavian Conference on Image Analysis, (SCIA)*, vol. 6688 of *Lecture Notes in Computer Science*, pp. 545–556, Springer, 2011.

[33] H. Knutsson, M. Andersson, and J. Wiklund, "Advanced filter design," in *Proceedings of the Scandinavian Conference on Image Analysis, (SCIA)*, pp. 185–193, 1999.

[34] H. Knutsson and C. F. Westin, "Normalized and differential convolution: methods for interpolation and filtering of incomplete and uncertain data," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, (CVPR)*, pp. 515–523, June 1993.

[35] Nvidia, *CUDA Programming Guide, Version 4.0.*, 2011.

[36] D. Kirk and W. Hwu, *Programming Massively Parallel Processors, A Handson Approach*, Morgan Kaufmann, Waltham, Mass, USA, 2010.

[37] The Khronos Group & OpenCL, 2010, http://www.khronos.org/opencl/.

[38] The OpenMP API specification for parallel programming, 2011, http://www.openmp.org/.

[39] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP, Portable Shared Memory Parallel Programming*, MIT Press, Cambridge, Mass, USA, 2007.

[40] V. W. Lee, C. Kim, J. Chhugani et al., "Debunking the 100X GPU vs. CPU Myth: an evaluation of throughput computing on CPU and GPU," in *Proceedings of the 37th International Symposium on Computer Architecture, (ISCA)*, pp. 451–460, June 2010.

[41] M. Andersson, J. Wiklund, and H. Knutsson, "Filter networks," in *Proceedings of the Signal and Image Processing, (SIP)*, pp. 213–217, 1999.

[42] B. Svensson, M. Andersson, and H. Knutsson, "Filter networks for efficient estimation of local 3-D structure," in *Proceedings of the IEEE International Conference on Image Processing, (ICIP)*, pp. 573–576, September 2005.

[43] M. Andersson, J. Wiklund, and H. Knutsson, "Sequential filter trees for efficient 2D, 3D and 4D orientation estimation," Tech. Rep. LiTH-ISY-R-2070, Department of Electrical Engineering, Linköping University, Linköping, Sweden, 1998.

[44] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, (SC)*, pp. 1–11, November 2008.

[45] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proceedings of the International Conference on High Performance Computing Networking, Storage and Analysis, (SC)*, pp. 1–10, November 2009.

[46] A. Sigfridsson, J. P. E. Kvitting, H. Knutsson, and L. Wigström, "Five-dimensional MRI incorporating simultaneous resolution of cardiac and respiratory phases for volumetric imaging," *Journal of Magnetic Resonance Imaging*, vol. 25, no. 1, pp. 113–121, 2007.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 481064, 9 pages
doi:10.1155/2011/481064

Research Article

Patient Specific Dosimetry Phantoms Using Multichannel LDDMM of the Whole Body

Daniel J. Tward,[1] Can Ceritoglu,[1] Anthony Kolasny,[1] Gregory M. Sturgeon,[2,3] W. Paul Segars,[2,4] Michael I. Miller,[1,5] and J. Tilak Ratnanather[1,5]

[1] *The Center for Imaging Science, The Johns Hopkins University, Baltimore, MD 21218-2686, USA*
[2] *Carl E. Ravin Advanced Imaging Laboratories, Duke University, Durham, NC 27705, USA*
[3] *Department of Biomedical Engineering, University of North Carolina, Chapel Hill, NC 27599-7575, USA*
[4] *Department of Radiology, Duke University Medical Center, Durham, NC 27710, USA*
[5] *Institute for Computational Medicine, The Johns Hopkins University, Baltimore, MD 21218-2686, USA*

Correspondence should be addressed to Daniel J. Tward, dtward@cis.jhu.edu

Received 1 April 2011; Accepted 3 June 2011

Academic Editor: Yasser M. Kadah

This paper describes an automated procedure for creating detailed patient-specific pediatric dosimetry phantoms from a small set of segmented organs in a child's CT scan. The algorithm involves full body mappings from adult template to pediatric images using multichannel large deformation diffeomorphic metric mapping (MC-LDDMM). The parallel implementation and performance of MC-LDDMM for this application is studied here for a sample of 4 pediatric patients, and from 1 to 24 processors. 93.84% of computation time is parallelized, and the efficiency of parallelization remains high until more than 8 processors are used. The performance of the algorithm was validated on a set of 24 male and 18 female pediatric patients. It was found to be accurate typically to within 1-2 voxels (2–4 mm) and robust across this large and variable data set.

1. Introduction

Measuring the radiation dose a patient accumulates through life is an important matter that has been receiving much attention recently, in particular for growing children (e.g., in the New England Journal of Medicine's recent critique of CT use [1], and the adoption of the Image Gently program [2] by the Society of Pediatric Radiology, the American Society of Radiologic Technologists, the American College of Radiology, the American Association of Physicists in Medicine, and others). While directly measuring dose to individual organs is impractical, the development of computational phantoms containing dosimetric information (e.g., [3]), such as the extended cardiac-torso (XCAT) phantom used in this study [4] have begun to be a reliable substitute. A key shortcoming of this strategy is that standard phantoms cannot adequately reflect variability between patients, especially for children of different sizes and ages, and defining new phantoms for each patient manually would be unfeasible. The strategy used here consists of manually segmenting a small subset of organs from pediatric CT data and calculating a full body mapping to a similarly segmented adult XCAT phantom [5]. The resulting transformation is used to map rich anatomical and dosimetric information to the child's body.

To map dense image data as well as point-based manifold data between adult and child, this application requires a smooth invertible transformation (a diffeomorphism) to be defined everywhere on the background space of the CT scan. Such transformations are an important focus of computational anatomy [6], where anatomical variability is understood by studying diffeomorphisms mapping anatomical manifolds to one another. Formally, anatomy is modelled as the quadruple $(\Omega, \mathcal{G}, \mathcal{I}, \mathcal{P})$, where Ω is the background space (i.e., subsets of $\mathbb{R}^3$), $\mathcal{G}$ is a group of diffeomorphisms on Ω, $\mathcal{I}$ is the orbit of a template I_0 under $\mathcal{G}$, and $\mathcal{P}$ is a family of probability measures on $\mathcal{G}$. Geodesic paths, $\phi_t \in \mathcal{G}$ for $t \in [0, 1]$, are used to evolve a template according to $I_0 \circ \phi_t^{-1}$, and a mapping to a target I_1 is defined when $I_1 = I_0 \circ \phi_1^{-1}$.

Large deformation diffeomorphic metric mapping (LDDMM) [7] generates such mappings ($\phi_1(x)$) by integrating a smooth time dependent velocity field $v_t(x)$ [8],

$$\phi_t(x) = \int_0^t v_{t'}(\phi_{t'}(x))dt', \tag{1}$$

with the initial condition being identity, $\phi_0(x) = x$. A functional of the velocity field, which enforces image matching as well as smoothness and ensures the path is a geodesic, is minimized as discussed below.

2. The Multichannel LDDMM Algorithm

There are existing algorithms for full body image registration, which are used (e.g.) in registering PET to CT data [9–11] and compensating for deformations such as breath holds. However, these tend to use elastic models, which are suitable for describing the small deformations that register two images of the same patient but are unable to accurately describe the widely varying deformations between adults and children of various ages. In addition to the constraints on smoothness and invertibility, transformations generated by LDDMM are well suited to this application, because its fluid model (rather than elastic) allows for large deformations to be generated [12] and because the submanifold preserving property of diffeomorphisms [13] allows a transformation calculated from a handful of segmented structures to be accurately applied to the thousands of anatomical structures defined in the XCAT phantom. Moreover, additional properties are well suited to future exploration. For example, LDDMM allows metric distances to be defined between template and target anatomies [8, 14] and allows statistical codification of anatomy [15, 16].

In this work, we use multichannel LDDMM (MC-LDDMM), an algorithm which treats each segmented organ as a separate image linked by a common background space [17] to calculate diffeomorphisms. This is accomplished by calculating the velocity field minimizing the energy functional

$$E = \int dt \|Lv_t\|_2^2 + \sum_{i=1}^{M} \frac{1}{\sigma_i^2} \left\| I_0^i \circ \phi_{t=1}^{-1} - I_1^i \right\|_2^2, \tag{2}$$

where I_1^i and I_0^i are the ith (out of M) channels (organs) of the target and template images, $\phi_{t=1}$ is a diffeomorphism generated by integrating the velocity field v_t from $t = 0$ to 1, and σ_i^2 describes the contribution of the ith channel to the overall energy. The operator $L = -\gamma Id + \alpha\nabla^2$, where $\gamma = -1$ is fixed and α is varied, Id is identity, and ∇^2 is the Laplacian operator, ensures smoothness of the velocity field and resulting deformations, with larger α corresponding to smoother deformations, and smaller α corresponding to more accurate transformations.

The energy gradient can be computed as [17]

$$\nabla_v E_t = 2v_t - K\left[\sum_{i=1}^{M} \frac{2}{\sigma_i^2} |D\phi_{t,1}| \nabla J_t^{0i} \left(J_t^{0i} - J_t^{1i}\right)\right], \tag{3}$$

where K is the operator inverse of $L^\dagger L$, $|\cdot|$ denotes determinant and D denotes the Jacobian. The transformation generated by integrating (1) from time $t' = s$ to time $t' = t$ is denoted $\phi_{s,t}$ (i.e., $\phi_{s,t} = \phi_t \circ \phi_s^{-1} = \phi_{t,s}^{-1}$). The quantity J_t^{0i} is the ith template channel transformed up to time t (i.e., $J_t^{0i} = I_0^i \circ \phi_t^{-1} = I_0^i \circ \phi_{t,0}$), J_t^{1i} is the ith target channel transformed backwards from time 1 to time t (i.e., $J_t^{1i} = I_1^i \circ \phi_{t,1}$), and ∇ is simply the spatial gradient.

It can be seen that the transformation and its inverse must be defined at all times, which was discretized here into 11 equally spaced time points from $t = 0$ to $t = 1$. Calculating this transformation from the velocity field is a large part of the computational load. Integration in time is performed using semi-Lagrangian advection, a technique used in numerical weather prediction [18]. We use an implicit method for numerical integration, with 3 iterations per voxel at each timestep.

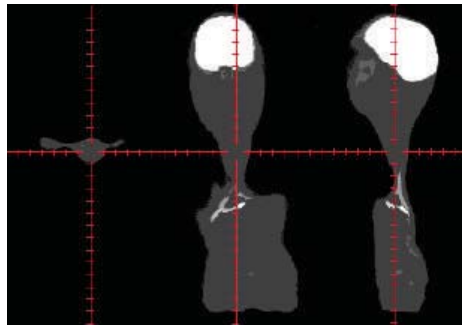
Moreover, a deformed target and template image must be calculated at each timestep. We use trilinear interpolation, which corresponds to another large computational load. To optimize calculations, the images for each channel were computed in the same loop (loop fusion).

Finally, application of the operator K is implemented by multiplication in the Fourier domain. The FFT calculations were performed and parallelized using Intel Math Kernel Library's (MKL) FFT routines.

Since many steps of this algorithm involve independent calculations on a regular 3D voxel grid, it is well suited to parallelization. In our C++ implementation of the LDDMM algorithm, OpenMP (open multiprocessing) library routines were used. As stated in [19], "the OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran on all architectures. ...OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface for developing parallel applications for platforms ranging from the desktop to the supercomputer." In our algorithm, at each iteration of gradient descent, different operations defined on data over the voxel grid were parallelized using work-sharing constructs, and loop iterations were split among the threads. The program was compiled using Intel C++ compiler version 12.0, with automatic compiler optimizations. It was run on a Dell R900, a 4 socket node with 6 cores per socket, with an Intel Xeon CPU E7450 at 2.40 GHz.

3. Methods

3.1. Calculation of Full Body Maps. In previous work [5], the feasibility of using multi-MC-LDDMM for this purpose was explored. A mapping to a single pediatric patient was calculated, and a reasonable subset of segmented organs was determined. However, generalizing this algorithm to a population of patients proved difficult. For example, where initial overlap of organs or bony details between template and target was poor, the diffeomorphism tended to shrink organs close to a point. Such distortions would also negatively affect the registration of nearby structures. Furthermore, when



(a) (b) (c)

Figure 1: An example of how the standard MC-LDDMM algorithm fails for full body mapping. (a) axial, (b) coronal, and (c) sagittal images of a deformed adult template. Notice that the abdominal organs have been catastrophically shrunk causing distortions in nearby neck and thoracic structures and that details in the face and skull have been lost.

structures were shrunk by the diffeomorphism details were lost, and when structures were expanded, their initial voxelized character was spuriously reproduced at the larger scale. These difficulties are illustrated by showing a deformed adult template in Figure 1, where abdominal organs are seen contracting to a very small size, nearby structures in the neck and thorax are distorted, and features in the face and skull are lost. Further investigation resulted in the algorithm being made more robust [20] but at the expense of increased computation time.

In the modified MC-LDDMM algorithm, (2) is minimized by initializing the velocity field to 0 and using a gradient descent routine with a large value of α. At convergence, the value of α is decreased, and minimization resumes, starting with the previously calculated velocity field. This procedure is iterated a total of four times. This sequential reduction of the parameter α (denoted "cascading α") allows for a coarse to fine registration and is responsible for the increased robustness as well as increased computation time of the modified algorithm. Beginning with a large value of α is analogous to Tikhonov regularization, encouraging a desirable solution to an ill posed problem. The final small value for α is chosen to give the desired level of accuracy in our mapping. Decreasing the value for α abruptly often results in nondiffeomorphic transformations due to numerical instability. So, we include 2 intermediate values to mitigate this effect and unfortunately must bear the price of considerably increasing computation time.

The MC-LDDMM algorithm with cascading α was used to generate mappings between one of two adult templates (one male and one female), and pediatric patients (24 male and 18 female). Each was defined on an $256\times256\times520$ 2mm^3 voxel grid. The patients varied in size between 0.072 and 0.472 times the volume of the adult, with an average of 0.233 times. Males ranged from 0.072 to 0.472 times the adult volume with a mean of 0.246, while females ranged from 0.076 to 0.372 times the adult volume with a mean of 0.215. The images were segmented into 8 channels with corresponding

Table 1: Segmented organs used for full body maps.

Organ	Weighting
Body	$\sigma_1 = 1$
Bones	$\sigma_2 = 1$
Kidneys	$\sigma_3 = 0.5$
Lungs	$\sigma_4 = 1$
Liver	$\sigma_5 = 1$
Spleen	$\sigma_6 = 0.5$
Stomach	$\sigma_7 = 0.5$
Brain	$\sigma_8 = 1$

organs and weightings defined in Table 1, and 87 landmarks were placed automatically [4] mainly on easily reproducible bony structures. Images were initially aligned with an affine transformation minimizing distances between corresponding landmarks, followed by nonlinear landmark LDDMM [21]. Following this, cascading α MC-LDDMM was used with the four values $\alpha = 0.05, 0.025, 0.01, 0.005$. In previous work, we found this particular sequence to give qualitatively good results in 2D simulations and 3D full body data [20].

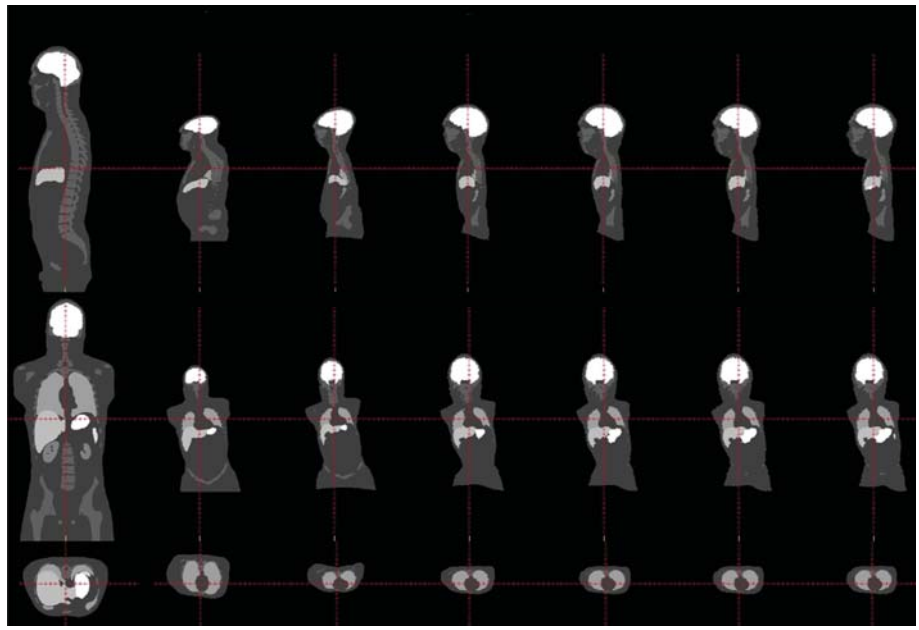
The sequence of transformations used to generate the final mapping is illustrated in Figure 2. Each transformations for each pediatric patient were combined to yield a double precision displacement vector at each voxel of the adult template images. This transformation was trilinearly interpolated to map NURBS (nonuniform rational B-spline) surfaces defined in the XCAT phantoms to the coordinate system of the child.

3.2. Analysis of Computation. The bulk of the computational work was performed during cascading α MC-LDDMM, and as such, its performance was investigated more thoroughly. Four patients were selected, 2 males and 2 females, corresponding to the largest, smallest, and 1/3 interquartile sizes, denoted "small", "med-small", "med-large", and "large". Mappings were calculated on these patients using each of 1, 2, 4, 8, 16, and 24 (the maximum readily available) processors. The total computation time excluding input-output (IO) operations was analyzed for each case as well as the time spent in specific functions. This allowed us an understanding of how computation time scales with the number of processors used, and in particular identify at what point computation time begins to increase beyond what would be expected.

To be more thorough, the portions of the program that were affected by parallelization, including IO operations, were analyzed. Speedup, c_n, due to parallelization on n processors was calculated (using "Amdahl's Law" [22] as in [23]) to be

$$c_n = \frac{T(1)}{T(n)} = \frac{A+B}{A+B/n}, \tag{4}$$

where $T(n)$ is the total computation time for n processors, and for a single processor, A is the time spent that cannot be



(a) (b) (c) (d) (e) (f) (g)

Figure 2: The robust sequence of transformations leading to the final mapping. Top row: sagittal slice, middle row: coronal slice, bottom row: axial slice. (a) Initial placement, (b) after affine registration, (c) after LDDMM landmark, and (d)–(g) after 1–4 iterations of MC-LDDMM.

Table 2: Summary of 4 subjects used to analyze computational performance.

Subject	Voxels	Iterations	~No. of calculations
Small	2459200	439	$1.08e+09$
Med-Small	6182224	942	$5.82e+09$
Med-Large	9358976	640	$5.99e+09$
Large	16082000	544	$8.75e+09$

parallelized, and B is the time spent that can be parallelized. These two quantities are easily estimated from a two parameter fit to the above equation, which allows determination of the fraction of the total computational time that can be parallelized. Furthermore, the efficiency of parallelization was calculated according to

$$e_n = \frac{c_n}{n}. \tag{5}$$

3.3. Accuracy of Mappings. Finally, the quality of the mappings produced was validated. For each segmented organ, a triangulated surface was produced using isosurface generation via marching tetrahedra [24]. For each template (target) vertex, the minimum distance to a vertex on the target (template) surface was measured. Distances for template and target vertices were combined, and their distributions were analyzed. Breaking down this analysis into categories allows an understanding of the robustness of the algorithm. As such, distributions were analyzed separately for males, females, as well as for each segmented organ.

4. Results

4.1. Computational Performance. A summary of the 4 subjects used to analyze computation performance is included in Table 2. The number of voxels in each image is shown in the second column, giving more precise meaning to the labels "small", "med-small", "med-large", and "large". The total number of iterations of gradient descent across the 4 applications of MC-LDDMM is shown in the third column. Due in part to adaptive stepsize selection in gradient descent, the number of iterations until convergence cannot be known before hand. In the fourth column, the product between number of voxels and number of iterations is shown as a rough approximation of the number of calculations used. This value can be used to better understand the timing results that follow. In particular the "med-small" case required the most iterations to converge, and the approximate number of calculations was much less for the "small" patient than for the other three. We stress that these four patients were chosen with interquartile spacing of their total number of voxels, as opposed to uniform spacing across number of voxels, or uniform spacing across number of calculations. Such a choice is reflective of the pediatric population to be examined, rather than properties of the algorithm itself.

The total computational time in hours, excluding IO operations, is shown in Table 3. The two largest components of calculations are also shown. Numerically integrating the

Table 3: Total timing (in hours) excluding IO operations.

Processors	Small	Med-small	Med-large	Large
1	8.94	33.5	31.3	28
2	4.9	18.2	17.3	15.2
4	2.62	9.68	9.05	7.92
8	1.49	5.41	5.07	4.47
16	1.06	3.64	3.5	3.1
24	0.935	3.25	3.17	2.8

Table 4: Semi-Lagrangian timing (in hours).

Processors	Small	Med-small	Med-large	Large
1	2.72	8.87	9.16	8.34
2	1.37	4.52	4.73	4.26
4	0.691	2.28	2.39	2.14
8	0.347	1.15	1.19	1.07
16	0.186	0.625	0.647	0.582
24	0.14	0.473	0.494	0.441

Table 5: Image interpolation timing (in hours).

Processors	Small	Med-small	Med-large	Large
1	2.28	8.27	9.24	7.91
2	1.25	4.59	5.07	4.3
4	0.653	2.45	2.63	2.21
8	0.352	1.29	1.38	1.16
16	0.251	0.869	0.88	0.771
24	0.219	0.776	0.767	0.685

velocity field using semi-Lagrangian interpolation is shown in Table 4, and trilinearly interpolating the images is shown in Table 5. Surprisingly, the longest amount of time was spent on the "med-small case". While this is partially explained by the large number of iterations for this case shown in Table 2, other factors such as the specific implementation of the fast Fourier transform on a grid of this size, contribute as well.

To better understand this behavior, the same data is shown graphically, on a log-log axes in Figure 3. Figure 3(a) shows the total time, Figure 3(b) shows the time spent calculating semi-Lagrangian advection, and Figure 3(c) shows the time spent interpolating images. It appears that computation time scales with number of processors up until around 8, when efficiency starts to break down.

Again, this data must be interpreted with caution, because the images used are different sizes and a different number of iterations of gradient descent is required to converge in each case, as shown in Table 2. Therefore, the timing data was also plotted after being normalized by total number of voxels times total number of iterations in Figure 4. It should be noted that the smallest image actually takes the most time per voxel per iteration, while the largest image takes the least.

The speedup factor and efficiency were calculated according to (4) and (5) and are plotted in Figure 5. This analysis confirms and quantifies the sharp drop in efficiency beyond 8 processors. From a 2 parameter fit to the data in Figure 5(a), it was determined that 93.84% of the computation time is parallelized, demonstrating the effectiveness of our implementation.

4.2. Accuracy of Transformations. To give a qualitative understanding of the mappings produced, an example of triangulated surfaces, for target and mapped template, are shown in Figure 6 with the body in Figure 6(a), the bones in Figure 6(b), and the other organs in Figure 6(c). One can see the quality of the mappings is good in most areas, the exceptions being the inferior-most regions, where the extent of template and target images vary, the scapula, where sliding motions between the nearby ribs and body surface are difficult to generate given the diffeomorphism constraint, and the sharp borders of some abdominal organs, whose curvature varies markedly from that of the template.

The mappings produced were used to generate customized dosimetry phantoms based on the adult XCATs. The adult male XCAT is shown in Figure 6(d) and an example pediatric dosimetry phantom is shown in Figure 6(e). Previous work has shown dosimetry measurements generated with these phantoms to agree within 10% percent of ground truth [5].

Cumulative distribution functions for final surface to surface distances are shown in Figure 7. They are shown for all patients pooled together as well as for males and females separately in Figure 7(a). The differences in accuracy, on average, between male and female patients is negligible. Additionally, distribution functions are shown for each organ in Figure 7(b). And they are shown for each of the 42 patients in Figure 7(c).

The results show that the majority of surfaces (a fraction of $1/e \sim 1$ standard deviation of the vertices) agree within 2–4 mm or 1-2 voxels. Moreover, accuracy for females tends to be more variable than that for males, likely due to larger differences in body proportions between child and adult. Surprisingly, the least accurate case, apparent in Figure 7(c), is an average seeming patient of intermediate size between the med-small and med-large test cases. Furthermore, differences in accuracy for each organ are observed, where the brain is matched with the most fidelity and the stomach followed by lungs with the least fidelity. While these differences are small when compared to the voxel size, it is worth noting that the relatively poor performance for the stomach was likely due to its internal location and close proximity with many other abdominal structures, and the relatively poor performance of the lungs was likely due to large differences in curvature between the adult and child at the apexes and inferior borders.

5. Conclusions

This work presented an interesting application of diffeomorphic image registration, generating pediatric patient specific detailed dosimetry phantoms, made feasible on large scale due to parallel computing. The need for parallelization

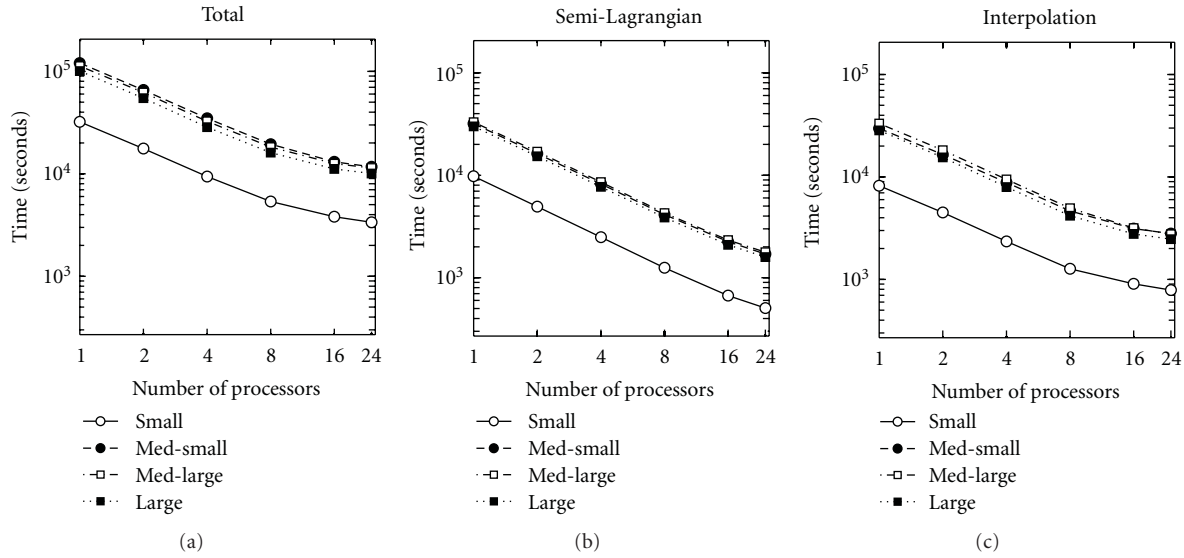


Figure 3: Time spent on computations for the four patients examined, plotted on a log-log axis. (a) Total time, (b) time in semi-Lagrangian advection, (c) time in image interpolation. Note that in (a) med-small takes the longest, followed by med-large, large, and small. In (b) and (c), the order of the first two is reversed.

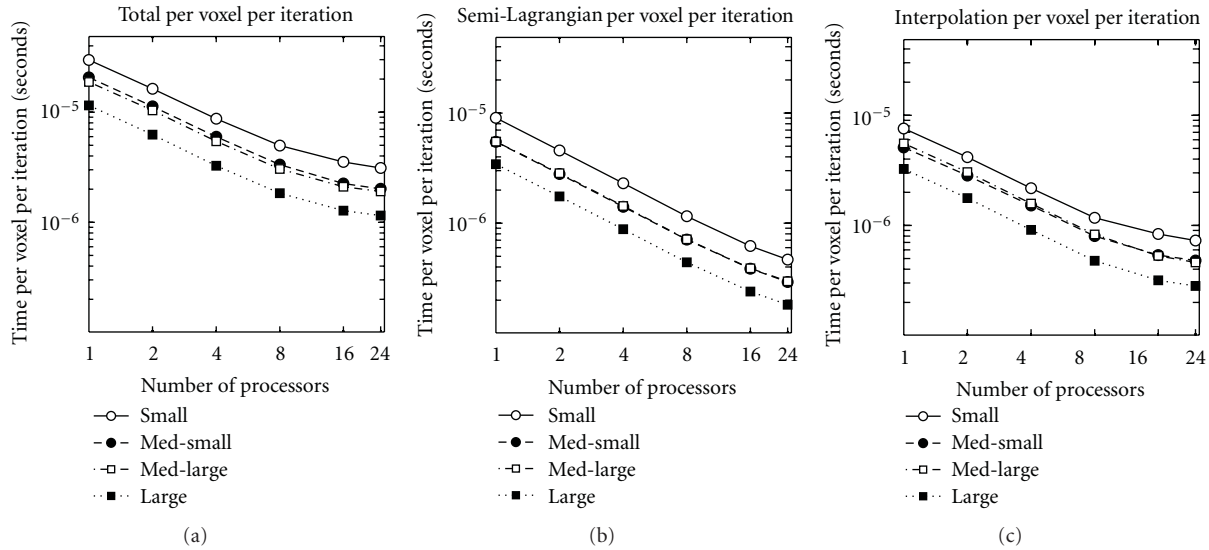


Figure 4: Time spent on computations, per image voxel per gradient descent iteration, for the four patients examined, plotted on a log-log axis. (a) Total time, (b) time in semi-Lagrangian advection, and (c) time in image interpolation. Note that in (a) small takes the longest, followed by med-small, med-large, and large. In (b) (for all processors) and (c) (from 1 to 8 processors), the order of the middle two is reversed.

in deformable image registration is well recognized [23, 25, 26], and other authors have investigated parallelization of diffeomorphic registration from MASPAR [27] to GPU implementations [28].

The algorithm used here for generating full body maps involves a sequence of increasingly detailed transformations between adult templates and child images. This procedure ensures robustness to automate calculations across a wide range of pediatric patients but comes at the price high computational cost.

To overcome this cost, 93.84% of the algorithm computation time was parallelized. Running times for the various patients examined ranged from over 30 hours on a single processor to under 1 hour on 24 processors in parallel. An analysis of speedup and parallelization efficiency shows that performance begins to rapidly decline when implemented on more than 8 processors. As applications for LDDMM become more numerous and larger scale, an investigation of this issue will be necessary. It is likely that the effects of memory to cpu communication bandwidth, load balancing

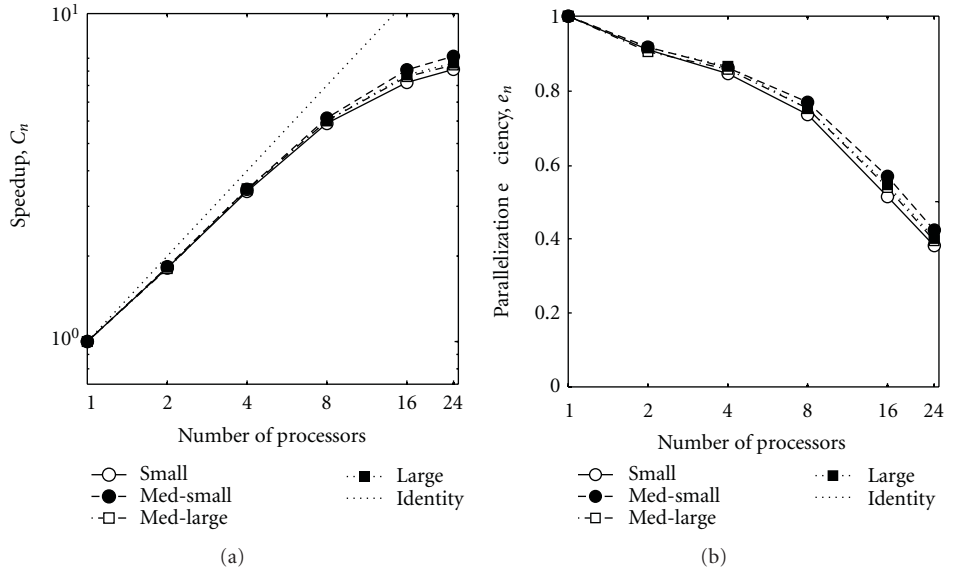


Figure 5: (a) Speedup due to parallelization (log scale) and (b) efficiency of parallelization (semilog scale), for the four patients examined. With the exception of "small" being uniformly the lowest, the order of the other varies as number of processors increases, and differences between each curve are quite small.

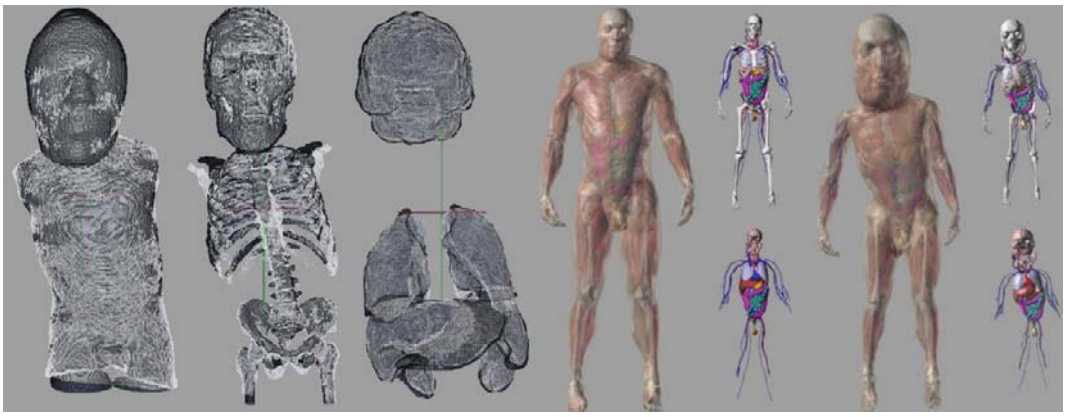


Figure 6: Triangulated surfaces from an example deformed adult template (white) and target child (black) are of (a) body, (b) bones, and (c) other organs. Adult male XCAT phantom is shown in (d), and an example custom dosimetry phantom is shown in (e).

overhead (due to workload not evenly distributed across the available processors) play a major role.

The full body mapping algorithm is quite accurate for all the patients examined, with the majority of vertices defined on organ surfaces agreeing between template and target to within 2 voxels. Overcoming a main drawback of the diffeomorphism constraint, namely, forbidding sliding motions in the deformation, is the subject of ongoing research. One strategy we are currently investigating involves relabelling a strip of the segmented image, between two structures where sliding would be expected, as "background". The XCAT phantoms generated are being further investigated for their accuracy and clinical utility.

While generating mappings using a sequence of transformations results in a robust algorithm for this application, it detracts from some of the theoretical appeal of LDDMM. Describing transformations by a single time-dependent vector field allows a rigorous study of anatomical variability. Future work will involve combining these transformations, for example, as described in [29], and beginning to engage in shape analysis of full bodies.

Acknowledgments

The authors would like to extend thanks to Joseph Hennessey for the development of a parallel visualization application

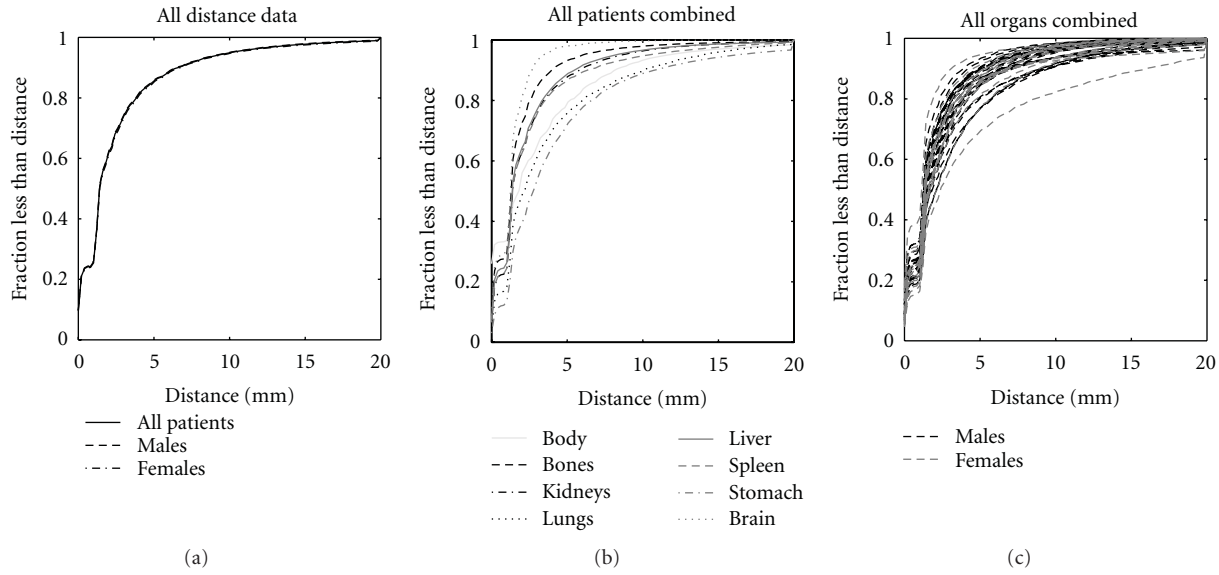


Figure 7: Cumulative distribution functions of final surface to surface distances are shown for all data and for all males and all females in (a), for individual organs with all patients combined in (b), and for individual patients with all organs combined in (c).

essential for this study, to Mike Bowers for developing and customizing an efficient parallelized implementation of landmark based LDDMM, and to Timothy Brown for assistance with computational infrastructure. D. J. Tward was supported by the the Julie-Payette NSERC Research Scholarship (Canada). The authors gratefully acknowledge the support of NIH Grants nos. 1S10RR025053-01, R01-EB001838, and P41-RR015241.

References

[1] D. J. Brenner and E. J. Hall, "Computed tomography—an increasing source of radiation exposure," *The New England Journal of Medicine*, vol. 357, no. 22, pp. 2277–2284, 2007.

[2] The Alliance for Radiation Safety in Pediatric Imaging, "Image gently.", 2009, http://www.pedrad.org/associations/5364/ig.

[3] C. Lee, D. Lodwick, J. L. Williams, and W. E. Bolch, "Hybrid computational phantoms of the 15-year male and female adolescent: applications to CT organ dosimetry for patients of variable morphometry," *Medical Physics*, vol. 35, no. 6, pp. 2366–2382, 2008.

[4] W. P. Segars, M. Mahesh, T. J. Beck, E. C. Frey, and B. M. W. Tsui, "Realistic CT simulation using the 4D XCAT phantom," *Medical Physics*, vol. 35, no. 8, pp. 3800–3808, 2008.

[5] W. P. Segars et al., "Patient specific computerized phantoms to estimate dose in pediatric CT," in *Proceedings of the Medical Imaging: Physics of Medical Imaging*, vol. 7258 of *Proceedings of SPIE*, p. 72580H, Lake Buena Vista, Fla, USA, February 2009.

[6] U. Grenander and M. I. Miller, "Computational anatomy: an emerging discipline," *Quarterly of Applied Mathematics*, vol. 56, no. 4, pp. 617–694, 1998.

[7] M. F. Beg, M. I. Miller, A. Trouvé, and L. Younes, "Computing large deformation metric mappings via geodesic flows of diffeomorphisms," *International Journal of Computer Vision*, vol. 61, no. 2, pp. 139–157, 2005.

[8] M. I. Miller, A. Trouvé, and L. Younes, "On the metrics and Euler-Lagrange equations of computational anatomy," *Annual Review of Biomedical Engineering*, vol. 4, pp. 375–405, 1998.

[9] R. Shekhar, V. Walimbe, S. Raja et al., "Automated 3-dimensional elastic registration of whole-body PET and CT from separate or combined scanners," *Journal of Nuclear Medicine*, vol. 46, no. 9, pp. 1488–1496, 2005.

[10] C. Cohade, M. Osman, L. T. Marshall, and R. L. Wahl, "PET-CT: accuracy of PET and CT spatial registration of lung lesions," *European Journal of Nuclear Medicine and Molecular Imaging*, vol. 30, no. 5, pp. 721–726, 2003.

[11] P. J. Slomka, D. Dey, C. Przetak, U. E. Aladl, and R. P. Baum, "Automated 3-Dimensional registration of stand-alone 18F-FDG whole-body PET with CT," *Journal of Nuclear Medicine*, vol. 44, no. 7, pp. 1156–1167, 2003.

[12] G. E. Christensen, R. D. Rabbitt, and M. I. Miller, "Deformable templates using large deformation kinematics," *IEEE Transactions on Image Processing*, vol. 5, no. 10, pp. 1435–1447, 1996.

[13] W. Boothby, *An introduction to Differentiable Manifolds and Riemannian Geometry*, Acedemic Press, 1986.

[14] M. I. Miller, A. Trouvé, and L. Younes, "Geodesic shooting for computational anatomy," *Journal of Mathematical Imaging and Vision*, vol. 24, no. 2, pp. 209–228, 2006.

[15] M. Vaillant, M. I. Miller, L. Younes, and A. Trouvé, "Statistics on diffeomorphisms via tangent space representations," *NeuroImage*, vol. 23, no. 1, pp. S161–S169, 2004.

[16] J. Ma, M. I. Miller, A. Trouvé, and L. Younes, "Bayesian template estimation in computational anatomy," *NeuroImage*, vol. 42, no. 1, pp. 252–261, 2008.

[17] C. Ceritoglu, K. Oishi, X. Li et al., "Multi-contrast large deformation diffeomorphic metric mapping for diffusion tensor imaging," *NeuroImage*, vol. 47, no. 2, pp. 618–627, 2009.

[18] A. Staniforth and J. Cote, "Semi-Lagrangian integration schemes for atmospheric models - a review," *Monthly Weather Review*, vol. 119, no. 9, pp. 2206–2223, 1991.

[19] OpenMP, "The OpenMP API specification for parallel programming," 2011, http://openmp.org/wp/.

[20] D. J. Tward, C. Ceritoglu, G. Sturgeon, W. P. Segars, M. I. Miller, and J. T. Ratnanather, "Generating patient-specific dosimetry phantoms with whole-body diffeomorphic image registration," in *Proceedings of the IEEE 37th Annual Northeast Bioengineering Conference*, Troy, NY, USA, April 2011.

[21] S. C. Joshi and M. I. Miller, "Landmark matching via large deformation diffeomorphisms," *IEEE Transactions on Image Processing*, vol. 9, no. 8, pp. 1357–1370, 2000.

[22] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," *Proceedings in AFIPS Converence*, vol. 7, pp. 483–485, 1967.

[23] T. Rohlfing and C. R. Maurer, "Nonrigid image registration in shared-memory multiprocessor environments with application to brains, breasts, and bees," *IEEE Transactions on Information Technology in Biomedicine*, vol. 7, no. 1, pp. 16–25, 2003.

[24] M. Joshi, J. Cui, K. Doolittle et al., "Brain segmentation and the generation of cortical surfaces," *NeuroImage*, vol. 9, no. 5, pp. 461–476, 1999.

[25] Y. Liu, A. Fedorov, R. Kikinis, and N. Chrisochoides, "Real-time non-rigid registration of medical images on a cooperative parallel architechture," in *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine*, pp. 401–404, Washington, DC, USA, November 2009.

[26] F. Ino, K. Ooyama, and K. Hagihara, "A data distributed parallel algorithm for nonrigid image registration," *Parallel Computing*, vol. 31, no. 1, pp. 19–43, 2005.

[27] G. E. Christensen, M. I. Miller, M. W. Vannier, and U. Grenander, "Individualizing neuroanatomical atlases using a massively parallel computer," *Computer*, vol. 29, no. 1, pp. 32–38, 1996.

[28] L. K. Ha, J. Krüuger, P. T. Fletcher, S. Joshi, and C. T. Silva, "Fast parallel unbiased diffeomorphic atlas construction on multi-graphics processing units," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, Munich, Germany, March 2009.

[29] L. Risser, F. Vialard, R. Wolz, M. Murgasova, D. Holm, and D. Rueckert, "Simultaneous multiscale registration using large deformation diffeomorphic metric mapping," *IEEE Transactions on Medical Imaging*. In press.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 698908, 12 pages
doi:10.1155/2011/698908

Research Article

CUDA-Accelerated Geodesic Ray-Tracing for Fiber Tracking

Evert van Aart,[1,2] Neda Sepasian,[1,2] Andrei Jalba,[1] and Anna Vilanova[2]

[1] *Department of Mathematics and Computer Science, Eindhoven University of Technology, 5600 MB Eindhove, The Netherlands*
[2] *Department of Biomedical Engineering, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands*

Correspondence should be addressed to Anna Vilanova, a.vilanova@tue.nl

Received 28 February 2011; Revised 17 June 2011; Accepted 24 June 2011

Academic Editor: Khaled Z. Abd-Elmoniem

Diffusion Tensor Imaging (DTI) allows to noninvasively measure the diffusion of water in fibrous tissue. By reconstructing the fibers from DTI data using a fiber-tracking algorithm, we can deduce the structure of the tissue. In this paper, we outline an approach to accelerating such a fiber-tracking algorithm using a Graphics Processing Unit (GPU). This algorithm, which is based on the calculation of geodesics, has shown promising results for both synthetic and real data, but is limited in its applicability by its high computational requirements. We present a solution which uses the parallelism offered by modern GPUs, in combination with the CUDA platform by NVIDIA, to significantly reduce the execution time of the fiber-tracking algorithm. Compared to a multithreaded CPU implementation of the same algorithm, our GPU mapping achieves a speedup factor of up to 40 times.

1. Introduction

Diffusion-Weighted Imaging (DWI) is a recent, noninvasive Magnetic Resonance Imaging (MRI) technique that allows the user to measure the diffusion of water molecules in a given direction. Diffusion Tensor Imaging (DTI) [1] describes the diffusion measured with DWI as a second-order tensor. DWI works on the knowledge that the diffusion of water molecules within biological tissue is influenced by the microscopic structure of the tissue. The theory of Brownian motion dictates that molecules within a uniform volume of water will diffuse randomly in all directions, that is, the diffusion is *isotropic*. However, in the presence of objects that hinder the diffusion of water in some specific directions, the diffusion will become *anisotropic*. In fibrous tissue, the diffusion of water will be large in the direction parallel to the fibers and small in perpendicular directions. Therefore, DWI data is used to deduce and analyze the structure of fibrous tissue, such as the white matter of the brain, and muscular tissue in the heart. DWI data has been used during the planning stages of neurosurgery [2], and in the diagnosis and treatment of certain diseases, such as Alzheimer's disease [3], multiple sclerosis [4], and strokes [5]. Since the tissue of the white matter is macroscopically homogeneous, other imaging techniques, such as T2-weighted MRI, are unable to detect the structure of the underlying fibers, making DWI uniquely suitable for in vivo inspection of white matter.

The process of using the measured diffusion to reconstruct the underlying fiber structure is called fiber tracking. Many different fiber tracking algorithms have been developed since the introduction of DTI. This paper focuses on an approach in which fibers are constructed by finding geodesics on a Riemannian manifold defined by the DTI data. This technique, called geodesic ray-tracing [6, 7], has several advantages over other ones, such as its relatively low sensitivity to measurement noise, and its ability to identify multiple solutions between two points, which makes it suitable for analysis of complex structures.

One of the largest downsides of this algorithm is that it is computationally expensive. Our goal is to overcome this problem by mapping the geodesic ray-tracing algorithm onto the highly parallel architecture of a Graphical Processing Unit (GPU), using the CUDA programming language. Since fibers can be computed independently of each other, the geodesic ray-tracing algorithm can be meaningfully parallelized. As a result, the running time can be reduced by a factor of up to 40, compared to a multithreaded CPU implementation. The paper describes the structure of the CUDA implementation, as well as the relevant design considerations.

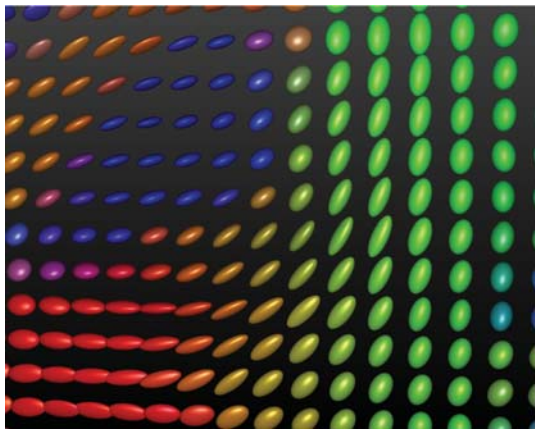


Figure 1: 3D glyphs visualizing diffusion tensors. The orientation and sharpness of the glyphs depend on the eigenvectors and eigenvalues of the diffusion tensor, respectively. In this image, the glyphs have been colored according to the orientation of the main eigenvector (e.g., a main eigenvector of $(1, 0, 0)$ corresponds to a red glyph, while a main eigenvector of $(0, 0, 1)$ corresponds to a blue glyph). This image was generated in the DTITool [11].

In the next section, we discuss the background theory related to our method, including DTI and the geodesic ray-tracing algorithm. Next, we give an overview of past research related to the GPU-based acceleration of fiber tracking algorithms. In Section 4, the implementation of the geodesic ray-tracing algorithm on a GPU using CUDA is discussed. Next, we show benchmarking results and optimization strategies for the CUDA implementation in Section 5, followed by a discussion of the results in Section 6, and a conclusion in Section 7.

2. Background

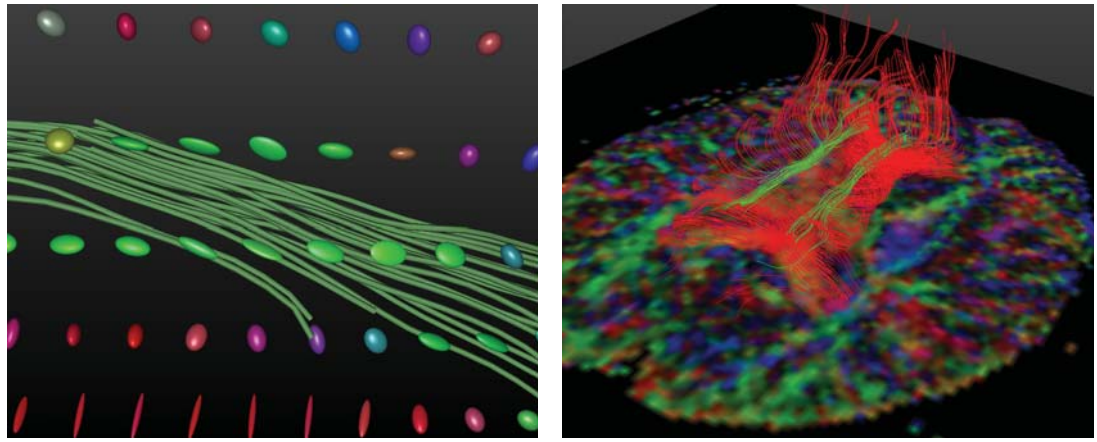
2.1. Diffusion Tensor Imaging. DTI allows us to reconstruct the connectivity of the white matter, thus giving us greater insight into the structure of the brain. After performing DWI for multiple different directions, we can model the diffusion process using a second-order tensor $\mathbf{D}$ [1, 8]. $\mathbf{D}$ is a 3×3 positive-definite tensor, which can be visualized as an ellipsoid defined by its eigenvectors and eigenvalues, as shown in Figure 1. Using the eigenvalues of a tensor, we can quantify its level of anisotropy using *anisotropy measures* [9, 10]. In areas with nearly isotropic diffusion, tensor ellipsoids will be nearly spherical, and anisotropy measure values will be low, while in areas with highly anisotropic diffusion (due to the presence of fibers), ellipsoids will be sharp and elongated, and anisotropy values will be high. In anisotropic areas, the eigenvector corresponding to the largest eigenvalue (the *main eigenvector*) will indicate the direction of the fibrous structure.

2.2. Fiber Tracking Algorithms. DTI Fiber Tracking is the process of digitally reconstructing the pathways of fibers in fibrous tissue using the DTI tensor data, with the aim of deducing the structure of the tissue. A common approach to fiber tracking is to track lines from one or more seed points, using a set of differential equations. The most straightforward fiber tracking algorithm (generally called the *streamline* method) uses the direction of the main eigenvector as the local orientation of the fibers [12]. Thus, a fiber in biological tissue may be reconstructed by integration of the main eigenvector, using an Ordinary Differential Equation (ODE) solver such as Euler's method. Figure 2 illustrates the relation between the diffusion tensors and the resulting fiber trajectories. Fiber tracking algorithms based on this approach have been shown to achieve acceptable results [13, 14], but are limited in their accuracy by a high sensitivity to noise and to the partial volume effect [15, 16].

One possible solution to the limitations of classic streamline methods is to use a global minimization solution, for example, a *front-propagation method*. In such a method, a front is propagated from a seed point throughout the entire volume [17–19]. The local propagation speed of the front depends on the characteristics of the DTI image, and fibers are constructed by back-tracing through the characteristics of the front. A subset of these front-propagation methods use the theory of geodesics to find potential fiber connections [19–21]. These algorithms generally compute the geodesics (defined as the shortest path through a tensor-warped space) by solving the stationary Hamilton-Jacobi (HJ) equation. Geodesic algorithms have been shown to produce good results and are generally more robust to noise than simple streamline methods. One disadvantage, however, is that they generally only find a single possible connection between target regions, which is often not the correct solution in very complex areas. Furthermore, research has shown it is possible to have multiple fiber connections between regions of the white matter [22].

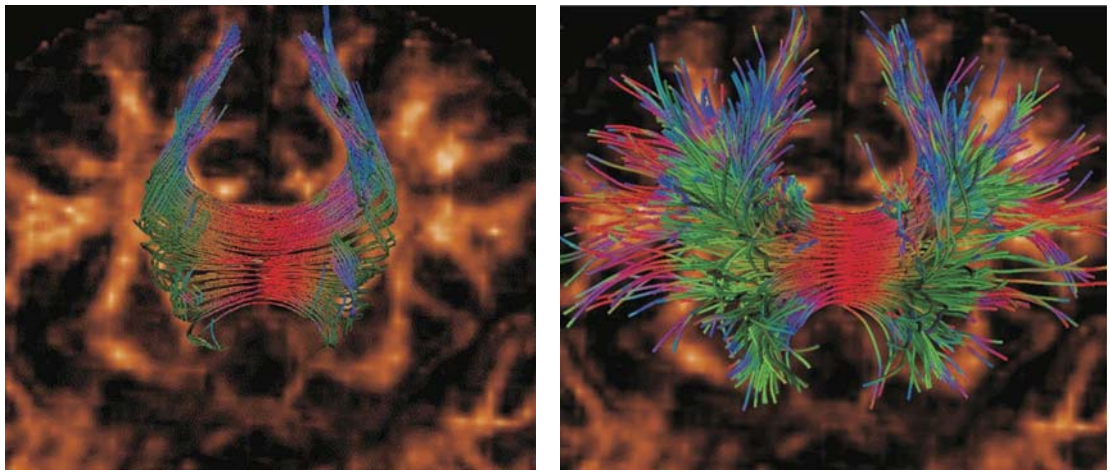
2.3. Geodesic Ray-Tracing. The focus of this paper is a relatively new fiber tracking algorithm based on geodesics, as proposed by Sepasian et al. [6, 7]. The main advantages of this algorithm, compared to those discussed in the previous section, are the fact that it provides a *multivalued* solution (i.e., it allows us to find multiple geodesics between regions in the brain), and the fact that it is able to detect fibers in regions with low anisotropy (e.g., regions of the white matter with crossing fiber bundles). In Figure 3, we show that this algorithm is capable of detecting complex structural features, such as the divergence of the corpus callosum, which cannot be captured using streamline-based methods. Detailed validation of this algorithm is considered beyond the scope of this paper; for an in-depth discussion on the validity and applicability of the algorithm, we refer the reader to the works by Sepasian et al. [6, 7].

A geodesic is defined as the shortest path on a Riemannian manifold. This is a real, differentiable manifold, on which each tangent space is equipped with a so-called *Riemannian metric*, which is a positive-definite tensor. Roughly speaking, the elements of the metric tensor are an indication of the *cost* of (or *energy* required for) moving in a specific direction. For DTI data, an intuitive choice for the metric is the *inverse* of the diffusion tensor. Large



(a) (b)

Figure 2: (a) Small group of fibers generated using a simple streamline method. The main eigenvectors of the diffusion tensors determine the local orientation of the fibers. (b) Fibers showing part of the Cingulum and the corpus callosum. Both the glyphs in the left image and the plane in the right image use coloring based on the direction of the main eigenvector, similar to Figure 1. Both images were generated in the DTITool [11].



(a) (b)

Figure 3: Fibers of part of the corpus callosum, computed using the streamlines method (a) and the geodesic ray-tracing method (b). Fibers were computed from seed points located in the center of the corpus callosum and are colored according to their local orientation (similar to the glyphs in Figure 1). Unlike the streamline method, which only captures the most dominant bundles of the corpus callosum, the geodesic ray-tracing method is able to correctly detect the divergence of the fiber bundles. This image was generated in the DTITool [11].

values in the DTI tensor correspond to small values in its inverse, indicating low diffusion costs, and vice versa. Locally, a geodesic will tend to follow the direction with the lowest metric value, which is analogous to the direction with the highest diffusion. We define the Riemannian metric as $G = D^{-1}$, where D is the diffusion tensor.

In the algorithm discussed in this paper, the trajectory of a fiber is computed iteratively by numerically solving a set of ODEs. The ODEs used to compute the trajectory of the fiber are derived from the theory of geodesics in a Riemannian manifold, as shown below.

Let $\mathbf{x}(\tau)$ be a smooth, differentiable curve through a volume described by parameter $\tau = [0, T]$, with derivative vector $\dot{\mathbf{x}}(\tau)$. We define the Riemannian length of $\mathbf{x}(\tau)$ as follows:

$$L(\mathbf{x}) = \int_0^T \sqrt{\dot{\mathbf{x}}^T G \dot{\mathbf{x}}} d\tau. \quad (1)$$

The geodesic is the line that *minimizes* the geodesic length of (1). We can use the Euler-Lagrange equations to translate this function to a set of ODEs, as described in detail by Jost [23].

Let $\dot{x}^\gamma$ and $\ddot{x}^\gamma$ be the first and second derivatives with regard to τ, respectively, of the geodesic for dimension

$\gamma = (1, 2, 3)$. The ODEs that allow us to compute the geodesics are given by the following equation:

$$\ddot{x}^{\gamma} + \sum_{\alpha=1}^{3}\sum_{\beta=1}^{3}\Gamma_{\alpha\beta}^{\gamma}\dot{x}^{\alpha}\dot{x}^{\beta} = 0, \tag{2}$$

where $\Gamma_{\alpha\beta}^{\gamma}$ are the so-called *Christoffel symbols*, defined as follows:

$$\Gamma_{\alpha\beta}^{\gamma} = \sum_{\sigma=1}^{3}\frac{1}{2}\left[g^{\gamma\sigma}\left(\frac{\partial}{\partial x_{\alpha}}g_{\beta\sigma} + \frac{\partial}{\partial x_{\beta}}g_{\alpha\sigma} - \frac{\partial}{\partial x_{\sigma}}g_{\alpha\beta}\right)\right]. \tag{3}$$

Here, g_{ij} represents element (i, j) of the inverse diffusion tensor, while g^{ij} represents an element of the *original* diffusion tensor. We note that, in order to compute all Christoffel symbols, we need to compute the derivatives of the inverse DTI tensor in all three dimensions.

Given an initial position and an initial direction, we can construct a path through the 3D DTI image, using the second-order Runge-Kutta ODE solver. The initial position is usually specified by the user, who is interested in a particular area of the tissue (in our case, the white matter). For the initial direction, we can use a large number of directions per seed points (distributed either uniformly on a sphere, or around the main eigenvector), and see which of the resulting fibers intersect some user-specified target region. Doing so increases our chances of finding all valid connections between the seed point(s) and the target region(s).

This approach, which is referred to as the Region-to-Region Connectivity approach, requires a suitable *Connectivity Measure* [24], which quantifies the strength of the connection between seed point and target region. In other words, this measure symbolizes the probability that a computed geodesic corresponds to an actual fibrous connection in the white matter. While this paper does not discuss the implementation of the Region-to-Region Connectivity approach, we do note that in order to reliably find all geodesics between the seed point(s) and the target region, we need to compute a large amount of trajectories in all directions. This need for a large amount of fibers, in combination with the high computational complexity of the algorithm itself, motivates our decision to parallelize the geodesic ray-tracing algorithm.

3. Related Work

The possibility of using the GPU to accelerate fiber tracking algorithms (using CUDA or other languages) has recently been explored in other literature [25–28]. These implementations use either geometric shaders or fragment shaders to accelerate the streamline tracking algorithm. With the exception of Mittmann et al. [28], who introduce a stochastic element, these papers all use the simple streamline method for fiber tracking, in which the main eigenvector of the DTI tensor is used as the direction vector for the integration step. In addition to using algorithms with a lower computational complexity than the geodesic ray-tracing algorithm discussed in Section 2.3, these implementations differ from ours in the sense that they use GPU shaders to compute the fibers, while we use CUDA, which offers higher flexibility and a more gentle learning curve than programmable GPU shaders [29].

More recently, Mittmann et al. introduced a GPU implementation of a simple streamline algorithm using CUDA [30]. Compared to a multithreaded CPU implementation, this GPU implementation allows for a significantly higher frame rate, enabling real-time, interactive exploration of large groups of fibers. The speedup factor, based on the number of frames per second, is between 10 and 20 times. The paper's main focus, however, is interactivity, and a technical discussion of the advantages and limitations of CUDA in the context of fiber tracking algorithms is omitted.

Jeong et al. [31] have developed a CUDA implementation of a fiber tracking algorithm based on the Hamilton-Jacobi equation, which computes the fiber pathways by propagating a front throughout the entire volume. Their solution parallelizes the propagation of the front by dividing the DTI image into blocks of 43 voxels, after which the front is propagated through a number of such blocks in parallel. This approach has been shown to be 50 to 100 times faster than sequential implementations of similar algorithms on a CPU. However, as stated in Section 2.3, the HJ algorithm on which they base their implementation is not able to find multiple connections between target regions. Furthermore, the front-propagation algorithm used by Jeong et al. requires a fundamentally different parallelization approach from our ray-tracing method.

We therefore conclude that our solution is the first CUDA-aided acceleration of a fiber tracking algorithm of this complexity. As will be shown in the next sections, the complex nature of the algorithm introduces a number of challenges to the parallelization process. Furthermore, the advantages of the geodesic ray-tracing algorithm, as discussed in Section 2.3, suggest that its implementation will also have practical applications.

4. Geodesic Fiber Tracking on the GPU Using CUDA

4.1. Algorithm Overview. In Section 2.3, we introduced a system of ODEs which can be used to compute the trajectory of a fiber, given an initial position and direction. The basis of our algorithm is an ODE solver which numerically solves (2) using a fixed integration step length. Specifically, let x_i^{γ} be the coordinate of point i along the fiber for dimension γ, and let $\dot{x}_i^{\gamma}$ be the local direction of the fiber in this point. Using Euler as the ODE solver, we can compute x_{i+1}^{γ} and $\dot{x}_{i+1}^{\gamma}$ (i.e., the position and direction at the next time step) as follows:

$$\begin{aligned} x_{i+1}^{\gamma} &= x_i^{\gamma} + h\dot{x}_i^{\gamma}, \\ \dot{x}_{i+1}^{\gamma} &= \dot{x}_i^{\gamma} - h\sum_{\alpha=1}^{3}\sum_{\beta=1}^{3}\Gamma_{\alpha\beta}^{\gamma}\dot{x}^{\alpha}\dot{x}^{\beta}. \end{aligned} \tag{4}$$

Here, h is a fixed step size, and $\Gamma_{\alpha\beta}^{\gamma}$ is the Christoffel symbol as defined in (3). In the implementation described below, we use a second-order Runge-Kutta ODE solver instead of

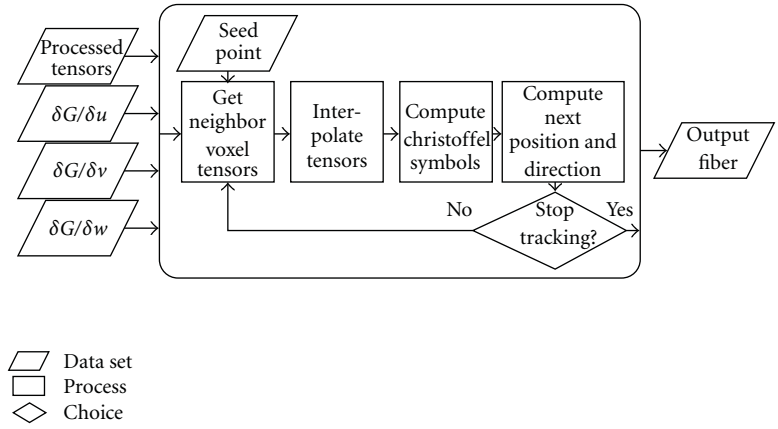


Figure 4: Flowchart for the geodesic fiber tracking algorithm. Using the four input tensor fields, we compute the trajectory of a fiber using a numerical ODE solver.

the Euler solver, but the basic method of numerically solving the system of ODEs remains the same. When computing the Christoffel symbols, the four required tensors (inverse DTI tensors and its three derivates) are interpolated using trilinear interpolation.

To summarize, a single integration step of the ODE solver consists of the following actions.

(1) Compute the inverse diffusion tensor and its derivative in all three dimensions for the eight voxels surrounding the current fiber point (x_i^y).

(2) Interpolate the four tensors at the current fiber point.

(3) Compute all Christoffel symbols. According to (2) and (3), we require nine symbols per dimension, for a total of 27 symbols. However, using the symmetric qualities of the diffusion tensor (and therefore, of its inverse and the derivates of its inverse), we can reduce this to 18 unique symbols.

(4) Using the Christoffel symbols, compute the position and direction of the next fiber point.

(5) Repeat steps 1 through 4 until some stopping condition is met. By default, the only stopping condition is the fiber leaving the volume of the DTI image.

We note that the first step may be performed as a pre-processing step. While this increases the amount of memory required by the algorithm, it also significantly decreases the number of required computations per integration step. This pre-processing step has been implemented in CUDA, but due to its trivial implementation and relatively low running time (compared the that of steps 2 through 5), we do not discuss it in detail, instead focusing on the actual tracking process. Figure 4 summarizes the tracking process of steps 2 through 5, assuming the four input tensor fields have been computed in a pre-processing step.

4.2. CUDA Overview. The Region-to-Region Connectivity approach outlined in Section 2.3 presents us with one significant problem: it requires the computation of a large number of geodesics. In combination with the relatively complex ODEs presented in (2) (compared to the ODEs used for simpler fiber tracking methods), this makes this approach computationally expensive. Our aim is to overcome this hurdle by implementing the algorithm in CUDA, of which we give a quick overview in this section.

NVIDIA's *Compute Unified Device Architecture* (CUDA) [32] is a way to facilitate General-Purpose computing on Graphics Processing Units (GPGPU). Modern GPUs contain large number of generic processors in parallel, and CUDA allows a programmer to utilize this large computational power for nongraphical purposes. In CUDA, a *kernel* (usually a small, simple function) is executed in parallel by a large number of *threads*, each working on one part of the input data. In a typical execution pattern, the host PC first uploads the input data to the GPU, then launches a number of threads that execute the kernel. The resulting data is then either downloaded back to the host PC or drawn on the screen.

A CUDA-enabled GPU generally consists of the *device memory*, which is between 512 MB and 2 GB on most modern GPUs, and a number of *multiprocessors* [33]. Each multiprocessor contains eight *scalar processors* (in most current-generation GPUs; newer generations will have more scalar processors per multiprocessor); a *register file*; a *shared memory* block, which enables communication between the scalar processors; and an *instruction unit*, which dispatches instructions to the processors. The type of parallelism in these multiprocessors is called *Single Instruction, Multiple Threads* (SIMT), which differs from Single Instruction, Multiple Data (SIMD) in the sense that the threads have some level of independence. Ideally, all active threads in a multiprocessor will execute the same instruction at the same time; however, unlike SIMD, SIMT also allows for branching threads using if-statements. While this does allow the programmer to create more complex kernels, it also adds an overhead in terms of execution time, since the different branches must be executed sequentially. Therefore, it is best to avoid branching behavior in kernels where possible.

One other important consideration when designing CUDA algorithms is the memory hierarchy. The two

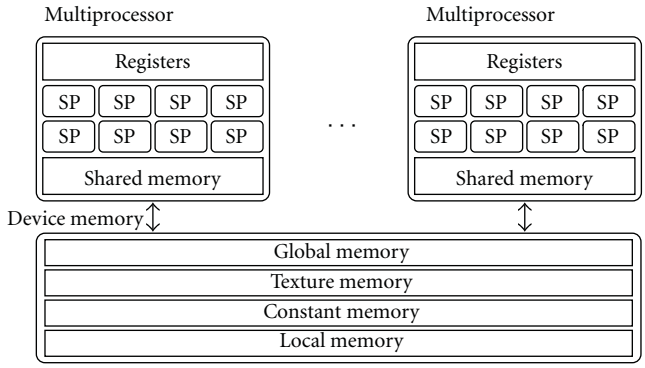


Figure 5: General structure of a CUDA-enabled GPU, showing the device memory, the multiprocessors, the scalar processors (SP), and the relevant memory spaces.

memory blocks local to each multiprocessor—the register file and the shared memory—both have low memory access latencies. However, the available space in these two memory blocks is limited, with typical values of 16 kB for the shared memory, and 16 or 32 kB for the register file. The device memory has far more storage space, but accessing this memory adds a latency of between 400 and 900 clock cycles [34]. The device memory contains four different memory spaces:

(i) *Constant Memory,* a small, read-only block best used for constant values;

(ii) *Texture Memory,* a read-only space optimized for texture reads;

(iii) *Global Memory,* the main random-access memory space;

(iv) *Local Memory,* a per-thread extension of the register file.

Local memory is used when the register file of a multiprocessor cannot contain all variables of a kernel. Since access latencies to the device memory are very high, the use of large kernels is generally best avoided. As a general rule, communication between the device memory and the multiprocessors should be kept as low as possible. A second important guideline is that the size of the kernels, in terms of the number of registers per thread and the amount of shared memory used per thread block, should be kept small. Doing so will allow the GPU to run more threads in parallel, thus increasing the *occupancy* of the scalar processors (i.e., the percentage of time that each scalar processor is active). The structure of a CUDA-enabled GPU is illustrated in Figure 5.

4.3. CUDA Implementation. Keeping in mind the advantages and limitations of CUDA as discussed in Section 4.2, we can design a CUDA implementation for the algorithm introduced in Section 2.3. As mentioned in step 3 of the algorithm as outlined in Section 4.1, we require the derivatives of the inverse of the DTI tensor in all three dimensions. We intuitively see that computing these derivatives for each point along the fiber would be far too costly in terms of the number of instructions, as computing these derivates for all eight surrounding voxels (using two-sided numerical derivation) would require the inversion of 32 diffusion tensors. Therefore, we decide to precompute them instead.

This gives us four input tensors per voxel: the diffusion tensor and the three derivatives of its inverse. With six unique elements per tensor and four bytes per value, this gives us a memory requirement of $4 * 6 * 4 = 96$ Bytes per voxel. The algorithm also has the seed points as input, which contain the initial position and direction for each fiber. These seed points are determined by the user, for example, by specifying a region of interest in the image.

The output of the algorithm consists of a number of fibers, each consisting of a list of 3D coordinates. In postprocessing steps, these fibers can be filtered through the target region(s) and sorted according to their Connectivity Measure value, but these steps are not part of the CUDA implementation discussed herein. Since CUDA does not support dynamic memory allocation, we hard-coded a limit in the number of iteration steps for each fiber, and statically allocated an output array per fiber of corresponding size. We note here that the choice of this limit may impact performance: for high limits, more fibers will terminate prematurely (due to leaving the volume), leading to lower occupancy in later stages of the algorithm, while for lower limits, the start-up overhead of the CUDA algorithm may become relevant.

We intuitively recognize two different approaches for parallelization of the geodesic ray-tracing algorithm: *per-region* parallelization and *per-fiber* parallelization. The per-region approach would entail loading a small region of the image into the shared memory of a multiprocessor, tracking a number of fibers through this region (using one thread per fiber), and then loading the next region. While this approach would guarantee low memory bandwidth requirements between the multiprocessors and the device memory, it is impractical due to two reasons. First, it is impossible to guarantee that a region will contain a sufficient number of fibers to enable meaningful parallelization. Second, due to the limited size of the shared memory, these regions would be very small (no more than approximately 160 voxels per region), which would defeat the purpose of this approach. In other words, this approach requires some degree of *spatial coherence* between the fibers, and since we do not know their trajectories beforehand, ensuring this spatial coherence is highly impractical.

We therefore use the per-fiber parallelization approach, in which each thread computes a single fiber. The main advantage of this approach is that it does not require any spatial coherence between the fibers to efficiently utilize the parallelism offered by the GPU. As long as we have a high number of seed points and initial directions, all scalar processors of the GPU will be able to run their own threads individual of the other threads, thus minimizing the need for elaborate synchronization between threads, and guaranteeing a stable computational throughput for all active processors. The main disadvantage of this approach is that it requires a higher memory throughput than the per-region approach, as it does not allow us to avoid redundant memory reads.

The kernel that executes the fiber tracking process executes the following steps, as summarized in Figure 4:

(1) Fetch the initial position and direction of the fiber.

(2) For the current fiber position, fetch the diffusion tensor and the derivatives of its inverse, using trilinear interpolation.

(3) Using these four tensors, compute the Christoffel symbols, as defined in (3).

(4) Using the symbols and the current direction, compute the next position and direction of the fiber, using the ODEs of (2) with a second-order Runge-Kutta step.

(5) Write the new position to the global memory.

(6) Stop if the fiber has left the volume *or* the maximum number of steps has been reached. Otherwise, return to Step 2.

4.4. Using Texture Memory. As noted in the previous section, the main disadvantage of the per-fiber parallelization approach is that it does not allow us to avoid redundant reads. We can partially solve this problem by storing the input images in *texture memory*, rather than in global memory. Unlike global memory reads, texture memory reads are cached through a number of small caches. These caches can contain about 8 kB per multiprocessor, although the actual amount varies per GPU. While we argue that cached memory reads could reduce the required memory throughput, we note that the lack of spatial coherence between the fibers, coupled with the small cache sizes, will largely negate the positive effects of cached memory reads.

A second, more important advantage of the texture memory is the built-in texture filtering functionality. When reading texture data from a position located between grid points, CUDA will apply either nearest-neighbor or linear interpolation using dedicated texture filtering hardware. When storing the input data in the global memory, all interpolation must be done *in-kernel*, which increases both the number of instructions per integration step, and the amount of memory required by the kernels. By delegating the interpolation process to this dedicated hardware, we are able to reduce both the size and the computational complexity of the kernels. The size is especially important in this case, as smaller kernels allow for a higher degree of parallelism.

While we expect the use of texture-filtering interpolation to be beneficial for the running time of our algorithm (which we will demonstrate in the next Section), we do identify one possible trade-off. One property of in-kernel interpolation is that the values of the eight surrounding voxels can be stored in either the local memory of the thread, or in the shared memory of the multiprocessor. Doing so increases the size of the threads, but also allows them to reuse some of this data, thus reducing the memory throughput requirements. Using texture-filtering interpolation, we cannot store the surrounding voxel values, so we need to read them again in every integration step. Thus, in-kernel interpolation may require a significantly lower memory throughput than texture-filtering interpolation, especially for small step sizes (in which case it takes multiple steps for a fiber to cross a cell). We analyze this trade-off through experimentation in the next section.

5. Experiments and Results

For the experiments presented in this section, we used a synthetic data set of $1024 \times 64 \times 64$ voxels with 2048 predefined seed points. The seed points were distributed randomly to mimic the low spatial coherence of the fibers, and their initial directions were chosen in such a way that no thread would terminate prematurely due to its fiber leaving the volume (i.e., all fibers stay within the volume for the predefined maximum number of integration steps, running parallel to the long edge of the volume). While neither the shape and content of the volume nor the fact that no fibers leave the volume prematurely can be considered realistic, this does allow us to benchmark the algorithm under maximum load.

All experiments were conducted on an NVIDIA GTX 260, a mid-range model with 24 multiprocessors (for a total of 192 scalar processors) and 1 GigaByte of device memory [35].

It should be noted that we only consider the actual fiber tracking process for our benchmarks. The data preparation stage (which includes preprocessing and inverting the tensors, and computing the derivatives of the inverse tensors) has also been implemented in CUDA, but is considered outside of the scope of this paper due to its low complexity, trivial parallelization, and low running times compared to the tracking stage. On the GTX 260, the data preparation stage requires roughly 50 milliseconds per million voxels [36], and only needs to be executed once per image. The overhead resulting from the communication between the CPU and GPU lies in the order of a few milliseconds.

5.1. Texture Filtering versus In-Kernel Filtering. In Section 4.4, we stated that using the dedicated texture filtering hardware for the trilinear interpolation step of our algorithm would significantly reduce the size and complexity of our kernel, allowing for an increase in performance. We also identified a possible trade-off: for small step sizes, in-kernel interpolation might be faster than texture-filtering interpolation, as the former allows for data reuse, while the latter does not. We test this hypothesis by varying the step size between 0.05 (i.e., twenty steps per cell on average) and 0.5 (two steps). The measured running times for the two proposed interpolation methods are shown in Figure 6. From this, we can conclude that, while small step sizes do indeed reduce the running times for in-kernel interpolation, texture-filtering interpolation is still the faster option for typical step size values (between 0.1 and 0.2).

5.2. Limiting Factor. The performance of CUDA programs is usually limited either by the maximum *memory throughput* between the device memory and the multiprocessors, or

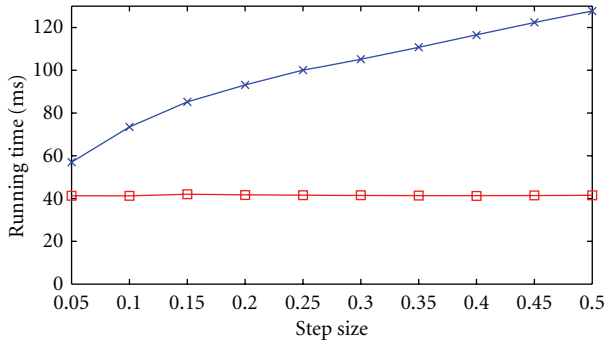


Figure 6: Running time for varying step size for in-kernel interpolation (blue) and texture-filtering interpolation (red).

by the maximum *computational throughput* of the scalar processors. To find out which of these two is the limiting factor for our algorithm, we first compute its performance in terms of memory throughput and computational throughput.

(i) When computing 2048 fibers, each executing 4096 integration steps, the algorithm takes approximately 165 milliseconds to complete. For each integration step, a single thread needs to read 8 voxels $*$ 4 tensors $*$ 6 unique tensor elements $*$ 4 bytes = 768 bytes, and it writes 12 bytes (3D coordinates of the new point). This gives us a total memory transfer of $780 * 2048 * 4096 \approx 6.54$ GB. Dividing this by the running time, we get an effective memory throughput of approximately 3.97 GB/s, which is well below the maximum 111.9 GB/s of the GTX 260.

(ii) To compute the computation throughput in FLOPS (floating-point operations per second), we first decompile the CUDA program using the decuda software. By inspecting the resulting code, we learn that each integration step uses 256 floating-point instructions. Note that this does not include any instructions needed for interpolation, since we are using the dedicated texture filtering hardware for this purpose. The algorithm performs a total of $256 * 2048 * 4096 = 2{,}147{,}483{,}648$ floating-point operations, giving us a computational throughput of roughly 13 GFLOPS. Again, this is well below the specified maximum of the GTX 260, which is 715 GFLOPS.

Since neither the memory throughput nor the computational throughput is close to the specified maximum, we need to determine the limiting factor in a different way. We first rule out the computational throughput as the limiting factor by replacing the current second-order Runge-Kutta (RK2) ODE solver by a simple Euler solver. This does not reduce the amount of data transferred between device memory and multiprocessors, but it does reduce the number of floating-point operations per integration step from 256 to 195. This, however, does not significantly impact the running time (165.133 ms for RK2 versus 165.347 ms for Euler), indicating that the computational throughput (i.e., processor load) is not a limiting factor in our case.

Table 1: Effects on the running time and total memory throughput of changing the amount of data read per voxel.

Data/Step (Byte)	Time (ms)	Bandwidth (GB/s)
768	165.133	38.8
640	134.124	40.0
512	103.143	41.6
384	90.554	35.6

To prove that the memory throughput is indeed the limiting factor, we need to reduce the amount of data read in each integration step, without changing the number of instructions. We do so by using the knowledge that the four input tensors of our synthetic data set share a number of duplicate values. By removing some of the redundant reads, we can reduce the memory requirements of each thread, without compromising the mathematical correctness of the algorithm. The results of this experiment are listed in Table 1. From this, we can conclude that the performance of our algorithm is limited by the memory throughput, despite the actual throughput being significantly lower than the theoretical maximum of the GTX 260. Possible explanations for this discrepancy are listed in Section 6.

5.3. Speed-Up Factor

5.3.1. Setup. To evaluate the performance of our CUDA implementation, we compare its running times to those of a C++ implementation of the same algorithm running on a modern CPU. In order to fully explore the performance gain, we use four different CUDA-supported GPUs: the Quadro FX 770M, the GeForce 8800 GT, the GeForce GTX 260, and the GeForce GTX 470. The important specifications of these GPUs are shown in Table 2. It should be noted that our CUDA implementation was developed for the GTX 260 and was not modified for execution on the other GPUs. In particular, this means that we did not optimize our algorithm to make use of the improved CUDA features of the more recent GTX 470 GPU.

The CPU running the C++ implementation is an Intel Core i5 750, which has four cores, with a clock speed of 2.67 GHz. In terms of both date of release and price segment, the i5 750 (released in fall 2009) is closest to the GTX 470 (spring 2010); at the time of writing, both are considered mid- to high-range consumer products of the previous generation.

For this benchmark, we use a brain DTI image with dimensions of $128 \times 128 \times 30$ voxels. Seed points are placed in a small, two-dimensional region of 22 by 4 voxels, located in a part of the corpus callosum. An approximation of the seed region, as well as the resulting fibers, is shown in Figure 8. Seed points are randomly placed within this region, with a random initial direction. The number of seed points varies from 1024 to 4096. This test thus closely

Table 2: Specifications of the GPUs in the benchmark test of Section 5.3. Source: http://www.nvidia.com/content/global/global.php.

	Device memory (MB)	Memory bandwidth (GB/s)	Number of scalar processors
Quadro FX 770M	512	25.6	32
GeForce 880 GT	512	57.6	112
GeForce GTX 267	896	111.9	192
GeForce GTX 470	1280	133.9	448

Table 3: Benchmark results for GPU and CPU implementation of the geodesic ray-tracing algorithm. For each configuration, we show the running time (T) in seconds, and the Speed-Up factor (SU) relative to the best CPU timing, see Figure 7.

	CPU	FX 770M		8800 GT		GTX 260		GTX 470	
Number of seeds	*T*	*T*	SU	*T*	SU	*T*	SU	*T*	SU
1024	1.403	0.761	1.8	0.273	5.1	0.225	6.2	0.087	16.1
2048	2.788	1.388	2.0	0.448	6.2	0.244	11.4	0.093	30.0
3072	4.185	1.995	2.1	0.760	5.5	0.256	16.3	0.107	39.1
4096	5.571	2.772	2.0	0.900	6.2	0.301	18.5	0.139	40.0

mimics a real-life application of the ray-tracing algorithm, as demonstrated in Figure 8.

5.3.2. CPU Implementation. Our C++ implementation of the algorithm features support for multiple threads of execution (multithreading), which allows it to utilize the parallelism offered by the CPU's four cores. Let S be the number of seed points and N the number of threads. We assign S/N seed points to each CPU thread, thus having each thread compute S/N fibers. We have measured the running times of this CPU implementation for several different values of N, and with S set to 4096 points. The results of this benchmark can be seen in Figure 7. From these results, we can conclude that parallelizing the CPU implementation (using essentially the same fiber-level parallelism as for our GPU implementation) can reduce the running times by a factor of roughly 4.5. The fact that the performance increases for N larger than the number of cores can be attributed to the fact that a CPU core can switch to a different thread when the active thread is waiting for data from the main memory of the PC. From Figure 7, we can also conclude that 64 threads is the best configuration for this algorithm and this CPU.

5.3.3. GPU Benchmark Results. We performed the same benchmarking experiment on the four CUDA-enabled GPUs, and we compared the running times to those of the best CPU configuration. The results for this test are shown in Table 3. From the results, we can conclude that our CUDA implementation significantly reduces the running time of the ray-tracing algorithm. Even on a mid-range GPU for laptops like the FX 770M, the running time is reduced by a factor of roughly two times. Using a high-end, recent GPU like the GTX 470, we are even able to achieve a speed-up factor of 40 times, which greatly increases the applicability of the algorithm. The differences in speed-up factors between the GPUs can be explained in part by the differences in bandwidth—which was identified as the limiting factor in Section 5.2—and by the number of processors.

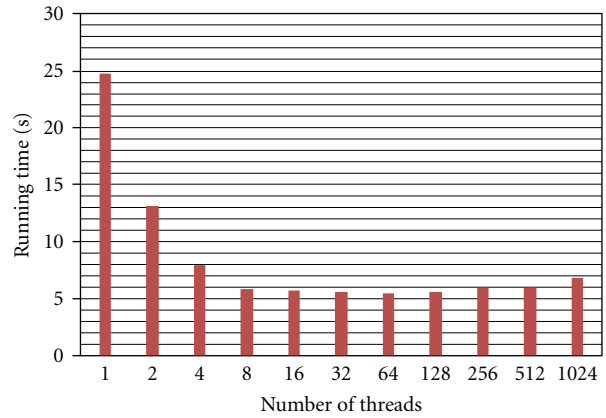


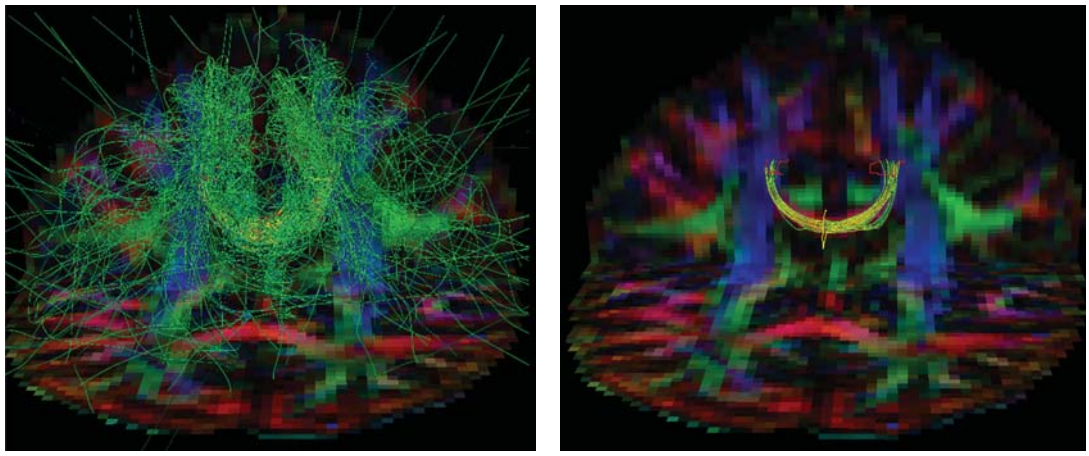
Figure 7: Running times (in seconds) for the multithreaded CPU implementation of our algorithm.

6. Discussion

In previous sections, we described a CUDA implementation of a geodesic fiber tracking method. This CUDA program uses the knowledge that fibers can be computed independently of one another to parallelize these computations, thus reducing running times by a factor of up to 40 times, compared to multithreaded CPU implementations of the same algorithm.

While the algorithm does allow for meaningful parallelization, we do note two problems that make full exploitation of the parallelism offered by the GPU challenging.

(i) The algorithm requires a large amount of internal storage; between the four input tensors, the Christoffel symbol, and the position and direction of the fiber, even the most efficient version of our implementation still required 43 registers per thread, while typical values for CUDA algorithms are between 8 and 32.



(a) (b)

Figure 8: Fibers computed using the CUDA ray-tracing algorithm in real data. (a) All fibers computed from a seeding region in part of the corpus callosum. (b) In order to properly analyze the computed fibers, a postprocessing step is needed. In this case, fibers were filtered through two target regions of interest and ranked and colored according to their Connectivity Measure value (see Section 2.3). The yellow polygon approximates the seeding region used for the benchmarks in Section 5.3. Both images were generated in the DTITool [11].

(ii) More importantly, the algorithm requires a large amount of data transfer, and as a result, the memory throughput becomes its limiting factor. This problem is far from trivial to solve, and while some options do exist, they all have their own downsides.

   (1) Reducing the number of bits per input value would lower the required memory throughput, but comes at the cost of reduced accuracy.
   (2) Allowing fibers to reuse the input data used for interpolation greatly increases the size of the kernel and does not work with texture-filtering interpolation, the use of which has been shown to be very beneficial in our case.
   (3) Sharing the input data between threads in one multiprocessor requires some sort of spatial coherence between the fibers, which is extremely difficult to guarantee.

As noted, the memory throughput between the device memory and the multiprocessors is the limiting factor for our performance. However, as noted in Section 5.2, the actual throughput is well below the theoretical maximum of the device. Below, we list some possible causes for this difference.

(iii) Most Random-Access Memory (RAM) configurations experience a certain overhead when subsequent reads access different parts of a data range. Since a low spatial locality of the seed points will lead to such scattered access pattern, this overhead may explain our relatively low memory throughput.

(iv) According to CUDA documentation [32], texture reads have been optimized for 2D spatial locality, presumably using a space-filling curve. The absence of spatial locality prevents our algorithm from utilizing these optimizations.

(v) The throughput of the texture fetching and filtering units may become a limiting factor when large numbers of voxels are involved. The documentation of the GTX 260 states that it should be able to process 36.9 billion texels per second [33], while our implementation only loads 39 billion bytes (of multibyte texels) per second. However, this figure is based on 2D bilinear interpolation, while we use 3D trilinear interpolation.

We expect the first point to be the main contributing factor, though we have not conducted experiments to either prove or disprove this hypothesis.

The GPU-based acceleration of the geodesic ray-tracing algorithm for DTI is a useful technique, but its implementation poses several challenges. Part of the problem in accelerating numerical integration algorithms such as the one under discussion in the paper is that it almost inevitably introduces unpredictable memory access patterns, while existing CUDA algorithms generally use access patterns that are more regular and predictable, and thus easier to optimize. This is a fundamental problem without an easy solution, and one that is not restricted to CUDA-enabled GPUs, but applies to other parallel platforms as well. Still, despite our implementation achieving suboptimal performance, we do believe that its significant speed-up factor of up to 40 times, coupled with the low cost and high availability of CUDA-enabled hardware, makes it a practical solution to the computational complexity of the geodesic ray-tracing algorithm for fiber tracking, and a good starting point for future acceleration of similar algorithms.

7. Conclusion and Future Work

In this paper, we discussed the GPU-based acceleration of a geodesic ray-tracing algorithm for fiber tracking for DTI.

One of the advantages of this algorithm is its ability to find multivalued solutions, that is, multiple possible connections between regions of the white matter. However, the high computational complexity of the algorithm, combined with the fact that we need to compute a large amount of trajectories if we want to find the right connection, makes it slow compared to similar algorithms. To overcome this problem, we accelerated the algorithm by implementing a highly parallel version on a GPU, using NVIDIA's CUDA platform. We showed that despite the large kernel size and high memory requirements of this GPU implementation, we were able to speed up the algorithm by a factor of up to 40. This significant reduction in running time using cheap and widely available hardware greatly increases the applicability of the algorithm.

Aside from further optimization of our current GPU implementation, with the aim of further reducing its running time, we identify two possible extensions of the work presented in this paper.

(i) At the moment, the computed fibers are downloaded back to the CPU, where they are postprocessed and subsequently visualized. A more direct approach would be to directly visualize the fibers computed by our algorithm, using the available rendering features of the GPU. If we can also implement the necessary postprocessing steps in CUDA, we can significantly reduce the amount of memory that needs to be transferred between the CPU and the GPU, thus accelerating the complete fiber tracking pipeline.

(ii) The Region-to-Region Connectivity method is a valuable application of the geodesic ray-tracing algorithm. This method introduces three extra steps: (1) computing a suitable Connectivity Measure, either while tracking the fibers or during a post-processing step, (2) filtering the computed fibers through a target region, and (3) sorting the remaining fibers according to their Connectivity Measure, showing only those fibers with high CM values. These three steps have currently been implemented on a CPU, but implementing them on a GPU can reduce the overall processing time of the fiber tracking pipeline, as noted above.

(iii) The DTI model is limited in its accuracy by its inability to model crossing fibers. High Angular Resolution Diffusion Imaging (HARDI) aims to overcome this limitation by measuring and modeling the diffusion in more directions [37]. An extension of the algorithm presented in Section 2.3 which uses HARDI data rather than DTI data would theoretically be more accurate, particularly in complex areas of the white matter. Such an extension may, for example, be realized by using the Finsler metric tensor (which depends both on the position and local orientation of the fiber), rather than the Riemannian metric tensor [38, 39]. While the extended algorithm for HARDI would likely be more complex in terms of both number of computations and amount of required memory, a CUDA-based acceleration using the parallelization principles presented in this paper will still be significantly faster than any CPU implementation.

We note that, while these extensions would be valuable additions, the current CUDA implementation already provides a practical and scalable solution for the acceleration of geodesic ray-tracing.

Acknowledgment

This project was supported by NWO project number 639.021.407.

References

[1] P. J. Basser, J. Mattiello, and D. Lebihan, "Estimation of the effective self-diffusion tensor from the NMR spin echo," *Journal of Magnetic Resonance, Series B*, vol. 103, no. 3, pp. 247–254, 1994.

[2] A. F. DaSilva, D. S. Tuch, M. R. Wiegell, and N. Hadjikhani, "A primer on diffusion tensor imaging of anatomical substructures," *Neurosurg Focus*, vol. 15, no. 1, p. E4, 2003.

[3] S. E. Rose, F. Chen, J. B. Chalk et al., "Loss of connectivity in Alzheimer's disease: an evaluation of white matter tract integrity with colour coded MR diffusion tensor imaging," *Journal of Neurology Neurosurgery and Psychiatry*, vol. 69, no. 4, pp. 528–530, 2000.

[4] M. Filippi, M. Cercignani, M. Inglese, M. A. Horsfield, and G. Comi, "Diffusion tensor magnetic resonance imaging in multiple sclerosis," *Neurology*, vol. 56, no. 3, pp. 304–311, 2001.

[5] D. K. Jones, D. Lythgoe, M. A. Horsfield, A. Simmons, S. C.R. Williams, and H. S. Markus, "Characterization of white matter damage in ischemic leukoaraiosis with diffusion tensor MRI," *Stroke*, vol. 30, no. 2, pp. 393–397, 1999.

[6] N. Sepasian, J. H. M. ten Thije Boonkkamp, A. Vilanova, and B. M. ter Haar Romeny, "Multi-valued geodesic based fiber tracking for diffusion tensor imaging," in *Proceedings of the Workshop on Diffusion Modelling and the Fiber Cup (MICCAI '09)*, pp. 6–13, 2009.

[7] N. Sepasian, A. Vilanova, L. Florack, and B. M. Ter Haar Romeny, "A ray tracing method for geodesic based tractography in diffusion tensor images," in *Proceedings of the Workshop on Mathematical Methods in Biomedical Image Analysis (MMBIA '08)*, 2008.

[8] C. G. Koay, L. -C. Chang, J. D. Carew, C. Pierpaoli, and P. J. Basser, "A unifying theoretical and algorithmic framework for least squares methods of estimation in diffusion tensor imaging," *Journal of Magnetic Resonance*, vol. 182, no. 1, pp. 115–125, 2006.

[9] P. J. Basser and C. Pierpaoli, "Microstructural and physiological features of tissues elucidated by quantitative-diffusion-tensor MRI," *Journal of Magnetic Resonance B*, vol. 111, no. 3, pp. 209–219, 1996.

[10] P. Van Gelderen, M. H.M. De Vleeschouwer, D. DesPres, J. Pekar, P. C.M. Van Zijl, and C. T.W. Moonen, "Water diffusion and acute stroke," *Magnetic Resonance in Medicine*, vol. 31, no. 2, pp. 154–163, 1994.

[11] Eindhoven University of Technology, Department of Biomedical Engineering, Biomedical Image Analysis Group, "DTITool," http://bmia.bmt.tue.nl/Software/DTITool.

[12] S. Wakana, H. Jiang, L. M. Nagae-Poetscher, P. C. M. Van Zijl, and S. Mori, "Fiber tract-based atlas of human white matter anatomy," *Radiology*, vol. 230, no. 1, pp. 77–87, 2004.

[13] P. J. Basser, S. Pajevic, C. Pierpaoli, J. Duda, and A. Aldroubi, "In vivo fiber tractography using DT-MRI data," *Magnetic Resonance in Medicine*, vol. 44, no. 4, pp. 625–632, 2000.

[14] T. E. Conturo, N. F. Lori, T. S. Cull et al., "Tracking neuronal fiber pathways in the living human brain," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 96, no. 18, pp. 10422–10427, 1999.

[15] M. Lazar, D. M. Weinstein, J. S. Tsuruda et al., "White matter tractography using diffusion tensor deflection," *Human Brain Mapping*, vol. 18, no. 4, pp. 306–321, 2003.

[16] C. Poupon, J. -F. Mangin, C. A. Clark et al., "Towards inference of human brain connectivity from MR diffusion tensor data," *Medical Image Analysis*, vol. 5, no. 1, pp. 1–15, 2001.

[17] M. Jackowski, C. Y. Kao, M. Qiu, R. T. Constable, and L. H. Staib, "White matter tractography by anisotropic wavefront evolution and diffusion tensor imaging," *Medical Image Analysis*, vol. 9, no. 5 SPEC. ISS., pp. 427–440, 2005.

[18] S. Jbabdi, P. Bellec, R. Toro, J. Daunizeau, M. Pélégrini-Issac, and H. Benali, "Accurate anisotropic fast marching for diffusion-based geodesic tractography," *International Journal of Biomedical Imaging*, vol. 2008, no. 1, 2008.

[19] C. Lenglet, R. Deriche, and O. Faugeras, "Inferring white matter geometry from diffusion tensor MRI: Application to connectivity mapping," *Lecture Notes in Computer Science*, vol. 3024, pp. 127–140, 2004.

[20] L. O'Donnell, S. Haker, and C. F. Westin, "New approaches to estimation of white matter connectivity in diffusion tensor MRI: elliptic PDEs and geodesics in a tensor-warped space," in *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI'02)*, vol. 5, pp. 459–466, 2002.

[21] E. Prados, C. Lenglet, J. -P. Pons et al., "Control theory and fast marching techniques for brain connectivity mapping," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, pp. 1076–1083, 2006.

[22] G. J. M. Parker, C. A. Wheeler-Kingshott, and G. J. Barker, "Distributed anatomical connectivity derived from diffusion tensor imaging," in *Proceedings of the Information Processing in Medical Imaging*, vol. 2082 of *Lecture Notes in Computer Science*, pp. 106–120, 2001.

[23] J. Jost, *Riemannian Geometry and Geometric Analysis*, Springer, New York, NY, USA, 5th edition, 2008.

[24] L. Astola, L. Florack, and B. M. ter Haar Romeny, "Measures for pathway analysis in brain white matter using diffusion tensor images," in *Proceedings of the Information processing in medical imaging*, vol. 4584 of *Lecture Notes in Computer Science*, pp. 642–649, 2007.

[25] A. Köhn, J. Klein, F. Weiler, and H. -O. Peitgen, "A GPU-based fiber tracking framework using geometry shaders," in *Proceedings of SPIE Medical Imaging*, vol. 7261, p. 72611J, 2009.

[26] P. Kondratieva, J. Krüger, and R. Westermann, "The application of GPU particle tracing to diffusion tensor field visualization," in *Proceedings of the IEEE Visualization Conference*, pp. 73–78, 2005.

[27] T. McGraw and M. Nadar, "Stochastic DT-MRI connectivity mapping on the GPU," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1504–1511, 2007.

[28] A. Mittmann, E. Comunello, and A. von Wangenheim, "Diffusion tensor fiber tracking on graphics processing units," *Computerized Medical Imaging and Graphics*, vol. 32, no. 7, pp. 521–530, 2008.

[29] T.-T. Wong, "Shader programming vs CUDA," in *Proceedings of the IEEE World Congress on Computational Intelligence*, 2008.

[30] A. Mittmann, T. H.C. Nobrega, E. Comunello et al., "Performing real-time interactive fiber tracking," *Journal of Digital Imaging*, vol. 24, no. 2, pp. 339–351, 2010.

[31] W.-K. Jeong, P. T. Fletcher, R. Tao, and R. T. Whitaker, "Interactive visualization of volumetric white matter connectivity in DT-MRI using a parallel-hardware Hamilton-Jacobi solver," in *Proceedings of the IEEE Visualization*, vol. 13, no. 6, pp. 1480–1487, November 2007.

[32] NVIDIA, "CUDA—programming guide," Version 2.3.1, 2009, http://www.nvidia.com.

[33] NVIDIA, "Technical brief—NVIDIA GeForce GTX 200 GPU architectural overview," 2008, http://www.nvidia.com.

[34] NVIDIA, "CUDA—C programming best practices guide," 2009, http://www.nvidia.com.

[35] NVIDIA, "GeForce GTX 260—specifications," 2010, http://www.nvidia.com.

[36] E. van Aart, *Acceleration of a geodesic fiber-tracking algorithm for diffusion tensor imaging using CUDA*, M.S. thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 2010.

[37] D. S. Tuch, T. G. Reese, M. R. Wiegell, N. Makris, J. W. Belliveau, and J. Van Wedeen, "High angular resolution diffusion imaging reveals intravoxel white matter fiber heterogeneity," *Magnetic Resonance in Medicine*, vol. 48, no. 4, pp. 577–582, 2002.

[38] L. Astola and L. Florack, "Finsler geometry on higher order tensor fields and applications to high angular resolution diffusion imaging," *Lecture Notes in Computer Science*, vol. 5567, pp. 224–234, 2009.

[39] L. Florack, E. Balmashnova, L. Astola, and E. Brunenberg, "A new tensorial framework for single-shell high angular resolution diffusion imaging," *Journal of Mathematical Imaging and Vision*, vol. 38, no. 3, pp. 171–181, 2010.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 473128, 11 pages
doi:10.1155/2011/473128

Research Article

High-Performance 3D Compressive Sensing MRI Reconstruction Using Many-Core Architectures

Daehyun Kim,[1] Joshua Trzasko,[2] Mikhail Smelyanskiy,[1] Clifton Haider,[2] Pradeep Dubey,[1] and Armando Manduca[2]

[1] *Parallel Computing Lab, Intel Corporation, 2200 Mission College Boulevard Santa Clara, CA 95054, USA*
[2] *The Center for Advanced Imaging Research, Mayo Clinic, 200 First Street SW, Rochester, MN 55905, USA*

Correspondence should be addressed to Armando Manduca, manduca.armando@mayo.edu

Received 30 March 2011; Accepted 3 June 2011

Academic Editor: Yasser M. Kadah

Compressive sensing (CS) describes how sparse signals can be accurately reconstructed from many fewer samples than required by the Nyquist criterion. Since MRI scan duration is proportional to the number of acquired samples, CS has been gaining significant attention in MRI. However, the computationally intensive nature of CS reconstructions has precluded their use in routine clinical practice. In this work, we investigate how different throughput-oriented architectures can benefit one CS algorithm and what levels of acceleration are feasible on different modern platforms. We demonstrate that a CUDA-based code running on an NVIDIA Tesla C2050 GPU can reconstruct a $256 \times 160 \times 80$ volume from an 8-channel acquisition in 19 seconds, which is in itself a significant improvement over the state of the art. We then show that Intel's Knights Ferry can perform the same 3D MRI reconstruction in only 12 seconds, bringing CS methods even closer to clinical viability.

1. Introduction and Motivation

Magnetic resonance imaging (MRI) is a noninvasive medical imaging modality commonly used to investigate soft tissues in the human body. Clinically, MRI is attractive as it offers flexibility, superior contrast resolution, and use of only nonionizing radiation. However, as the duration of a scan is directly proportional to the number of investigated spectral indices, obtaining high-resolution images under standard acquisition and reconstruction protocols can require a significant amount of time. Prolonged scan duration poses a number of challenges in a clinical setting. For example, during long examinations, patients often exhibit involuntary (e.g., respiration) and/or voluntary motion (e.g., active response to discomfort), both of which can impart spatial blurring that may compromise diagnosis. Also, high temporal resolution is often needed to accurately depict physiological processes. Under standard imaging protocols, spatial resolution must unfortunately be sacrificed to permit quicker scan termination or more frequent temporal updates.

Rather than executing a low spatial resolution exam, contemporary MRI protocols often acquire only a subset of the samples associated with a high-resolution exam and attempt to recover the image using alternative reconstruction methods such as homodyne detection [1] or compressive sensing (CS). CS theory asserts that the number of samples needed to form an accurate approximation of an image is largely determined by the image's underlying complexity [2, 3]. Thus, if there exists a means of transforming the image into a more efficient (i.e., sparse or compressible) representation, less time may actually be required to collect the data set needed to form the high-resolution image [4].

Background-suppressed contrast-enhanced MR angiography (CE-MRA) is a very natural clinical target for CS methods. As the diagnosis of many conditions like peripheral vascular disease are based on both vessel morphology and hemodynamics, high spatial and temporal resolution images are consequently needed. CS enables the acquisition of all of this information in a single exam. Although several authors (e.g., [5–8]) have successfully demonstrated the application of CS methods to CE-MRA, the computationally intensive nature of these applications has so far precluded their clinical viability (e.g., published CS reconstruction times for a single 3D volume (CE-MRA or not) are often on the order of

hours [4, 7, 9–11]). As the results of a CE-MRA exam are often needed as soon as the acquisition completes (either for immediate clinical intervention or to guide additional scans), it is not practical to wait for the result of any currently implemented CS reconstructions. Instead, linear or other noniterative reconstructions that can be executed online (e.g., [12]) must be used even if they provide suboptimal results.

With the goal of reducing CS+MRI reconstruction times to clinically practical levels, several authors have recently considered the use of advanced hardware environments for their reconstruction implementations. Most of these techniques have focussed on the reconstruction of MRI data acquired using phased-array (i.e., multicoil) receivers, as this is the dominant acquisition strategy used in clinical practice. Chang and Ji [13, 14] considered a coil-by-coil approach to reconstructing phased-array MRI data. Although this strategy leads to natural task parallelization, with each element of a multicore processor independently handling the reconstruction on one coil image, they only demonstrated reconstruction times on the order of minutes, per 2D slice which is not clinically viable. Moreover, disjoint reconstruction of phased-array MRI data is well known to exhibit suboptimal performance when compared against joint reconstruction strategies like SENSE [15] and GRAPPA [16] and so, this strategy is of limited utility. Murphy et al. [17] later demonstrated that the SPIRiT reconstruction algorithm [18], which is a generalization of GRAPPA [16], can be significantly accelerated using graphics processors. They generated high-quality parallel image reconstructions in on the order of minutes per 3D volume, representing a significant advance towards clinical feasibility. More recently, Trzasko et al. [7, 8, 19] demonstrated CS reconstructions of time-resolved 3D CE-MRA images acquired using parallel imaging and a state-of-the-art Cartesian acquisition with a projection-reconstruction-like sampling (CAPR) strategy [12] also in a matter of only minutes per 3D volume using an advanced code implementation on a cluster system [20]. Their algorithm, which is essentially a generalization of SENSE [15] that employs auxiliary sparsity penalties and an efficient inexact quasi-Newton solver, was demonstrated to yield high quality reconstruction of 3D CE-MRA data acquired at acceleration rates upwards of 50x.

In this paper, we focus on Trzasko et al.'s CS+MRI reconstruction strategy for 3D CE-MRA and investigate the development, optimization, and performance analysis on several modern parallel architectures, including the latest quad- and six-core CPUs, NVIDIA GPUs, and Intel Many Integrated Core Architecture (Intel MIC). Our optimized implementation on a dual-socket six-core CPU is able to reconstruct a $256 \times 160 \times 80$ volume of the neurovasculature from an 8-channel, 10x accelerated (i.e., 90% undersampled) data set within 35 seconds, which is more than a 3x improvement over other conventional implementations [8, 19]. Furthermore, we show that our CS implementation scales very well to the larger number of cores in today's throughput-oriented architectures. Our NVIDIA Tesla C2050 implementation reconstructs the same dataset in 19 seconds, while our Intel's Knights Ferry further reduces the reconstruction time to 12 seconds, which is considered clinically viable. Finally, our research simulator shows that the reconstruction can be done in 6 seconds on 128 cores, suggesting that many-core architectures are a promising platform for CS reconstruction.

2. Methods

2.1. Acquisition and Recovery of CE-MRA Images. Following [12], CAPR adopts a SENSE-type [15] parallel imaging strategy. As such, the targeted data acquisition process for one time frame can be modeled as

$$\begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_C \end{bmatrix} = \begin{bmatrix} \Phi\mathcal{F}\Gamma_1 \\ \Phi\mathcal{F}\Gamma_2 \\ \vdots \\ \Phi\mathcal{F}\Gamma_C \end{bmatrix} f + n, \tag{1}$$

where f is a discrete approximation of the underlying image of interest, Γ_c is the cth coil sensitivity function, $\mathcal{F}$ is the 3D discrete Fourier transform (DFT) operator, Φ is a (binary) sampling operator that selects a prescribed subset of k-space values, n is complex additive white Gaussian noise (AWGN), and g_c is signal observed by the cth coil sensor. As described in [19], raw CAPR k-space data is background-subtracted and view-shared prior to execution of the CS reconstruction procedure. We let $h_c(t)$ denote the result after such preprocessing of g_c.

It was demonstrated in [8, 19] that background-suppressed CE-MRA images acquired by systems of the form in (1) can be accurately recovered by (approximately) solving the following unconstrained optimization problem:

$$\tilde{v} = \arg\min_{v} J(v), \tag{2}$$

where the cost functional

$$J(v) = \alpha \sum_{n\in\eta} P(D_n v) + \sum_{c=1}^{C} \|\Phi\mathcal{F}\Gamma_c v - h_c\|_2^2, \tag{3}$$

D_n is the finite spatial difference operator for some offset direction n (in the neighborhood η), and the penalty functional

$$P(v) = \sum_{x\in\Omega} \rho(v(x)), \tag{4}$$

for some concave metric functional $\rho(\cdot)$ [21]. Following [7], the nonconvex Laplace functional

$$\rho(\cdot) = 1 - \exp\left(\sigma^{-1}|\cdot|\right), \quad \sigma \in [0, \infty) \tag{5}$$

is herein adopted. Although not considered here, the ℓ_1-norm ($\rho(\cdot) = |\cdot|$) could also be used if so desired.

2.2. Numerical Optimization. In [7], an efficient inexact quasi-Newton algorithm was proposed for (approximately) solving (2) to reconstruct CAPR CE-MRA images. For completeness, this algorithm is briefly reviewed. Recall that

complex quasi-Newton iterations [22] are typically of the form

$$v_{i+1} = v_i - B^{-1}(v_i)L(v_i), \tag{6}$$

where the gradient of $J(v)$ (taken with respect to $\overline{v}$ [23]) and $B(\cdot)$ is an approximation of the complex Hessian of $J(v)$. The term "inexact" arises when Δ_i is only approximately determined, such as via truncated conjugate gradient (CG) iteration. More specifically, given (3),

$$\begin{aligned} L(v_i) = {} & \alpha \sum_{n\in\eta} D_n^* \Lambda(D_n v_i) D_n v_i \\ & + \sum_{c=1}^{C} \Gamma_c^* \mathcal{F}^* \Phi^* (\Phi \mathcal{F} \Gamma_c v_i - k_c(t)), \end{aligned} \tag{7}$$

where the ($\epsilon > 0$ smoothed) diagonal operator

$$\Lambda(D_n v_i)_{(x,x)} = \frac{1}{2|[D_n v_i](x)|_{\epsilon_i}} \cdot \frac{\partial \rho\left(|[D_n v_i](x)|_{\epsilon_i}\right)}{\partial |[D_n v_i](x)|_{\epsilon_i}}. \tag{8}$$

In their work, Trzasko et al. [8, 19] adopted the following analytical linear Hessian approximation:

$$B(v_i) = \frac{\alpha}{2} \sum_{n\in\eta} D_n^* \Lambda(D_n v_i) D_n + \sum_{c=1}^{C} \Gamma_c^* \mathcal{F}^* \Phi^* \Phi \mathcal{F} \Gamma_c, \tag{9}$$

which can be considered a generalization of Vogel and Oman's "lagged diffusivity" model [24] for total variation (TV) denoising and deblurring. For improved convergence, decreasing continuation is also performed on the functional smoothing parameter, ϵ [25].

In [19], an efficient C++ implementation of the above algorithm employing the templated class framework described by Borisch et al. [20] and both the MPI (http://www-unix.mcs.anl.gov/mpi/) and OpenMP (http://www.openmp.org/) libraries was described and executed on an 8-node dedicated reconstruction cluster, where each node had two 3.4 GHz Intel Xeon processors and 16 GB memory. For a single $256 \times 160 \times 80$ head volume reconstruction from 8-channel data and only 6 difference neighbors, reconstruction times of slightly less than 2 minutes were reported. Although these times represent a significant advancement over other existing works, they are still too long for routine clinical use.

3. Experiments

We used five datasets (two artificial and three clinical) to analyze the CS performance on three platforms: Intel CPUs, NVIDIA GPUs, and Intel MIC.

3.1. Experimental Data and Reconstruction Specifications. Five datasets are used for our experiments, whose volume size and memory footprint are: ($256 \times 64 \times 64$, 224 MB), ($256 \times 160 \times 32$, 280 MB), ($256 \times 160 \times 80$, 700 MB), ($256 \times 160 \times 84$, 735 MB), and ($256 \times 160 \times 88$, 770 MB), in the order of dataset 1 to 5, respectively. Datasets 1 and 2 were artificially generated, Dataset 3 represents a noncontrast-enhanced brain, and Datasets 4 and 5 represent contrast-enhanced vasculature. All MRI data were acquired on a 3T GE Signa scanner (v.20) using an 8-channel head array using the CAPR acquisition sequence. Prior to reconstruction, view sharing was performed on datasets 3–5, and background reference subtraction on dataset 4 and 5 as described in [12]. For all experiments, 5 outer and 15 inner (CG) iterations were executed under $W = 1$ (corresponding to 26 finite difference neighbors). ϵ-continuation (0.1 reduction) was performed at each outer iteration. Figure 1 shows the current clinical [12] and CS-type reconstruction [19] results for dataset 5. The CS reconstruction results for all versions optimized for different architectures discussed below were visually identical to that shown here. All architectures are compliant with IEEE single-precision floating-piont arithmetic standard [26].

3.2. Computing Architectures

Intel Core i7 Processor. The Intel Core i7 processor is an ×86-based multicore architecture which provides four/six cores (731 M/1.17 B transistors) on the same die. It features a superscalar out-of-order core supporting 2-way hyper-threading and 4-wide SIMD. Each core is backed by 32 KB L1 and 256 KB L2 caches, and all cores share an 8 MB/12 MB L3 cache. Quad and six-core CPUs provide 100 Gflops and 135 Gflops of peak single-precision computation respectively, as well 32 GB/s of peak memory bandwidth. To optimize CS on Core i7 processors, we took advantage of its SSE4 instructions using the Intel ICC auto-vectorizing compiler as well as hand-vectorization intrinsics. We parallelized the code with OpenMP and adopted a highly optimized FFT implementation from Intel's Math Kernel Library (*MKL*) 10.2.

NVIDIA Tesla. The NVIDIA Tesla C2050 [27, 28] (3 B transistors) provides 14 multiprocessors, each with 32 scalar processing units that share 128 KB of registers and a 64 KB on-chip memory. 32 scalar units are broken into two groups, where each group runs in lockstep. It features a hardware multithreading, which allows hundreds of thread contexts running concurrently to hide memory latency. All multiprocessors share a 768 KB L2 cache. The on-chip memory is software configurable, and it can be split into a 48 KB cache and a 16 KB shared memory space, or vice versa. Its peak single-precision computing performance is about 1.03 Tflops and its on-board GDDR memory provides up to 144 GB/s bandwidth. We used the CUDA [29] programming environment to implement CS on Tesla C2050. CUDA allows programmers to write a scalar program that is automatically organized into thread blocks to be run on multiprocessors. CUDA provides an open source FFT library (CUFFT 3.1 [30]) although more optimized FFT implementations such as Nukada and Matsouka's [31] have been published.

3.2.1. Intel's Knights Ferry. Intel MIC is an Aubrey Isle- [32] based platform and Intel's Knights Ferry [33] is its first silicon implementation with 32 cores running at 1.2 GHz.

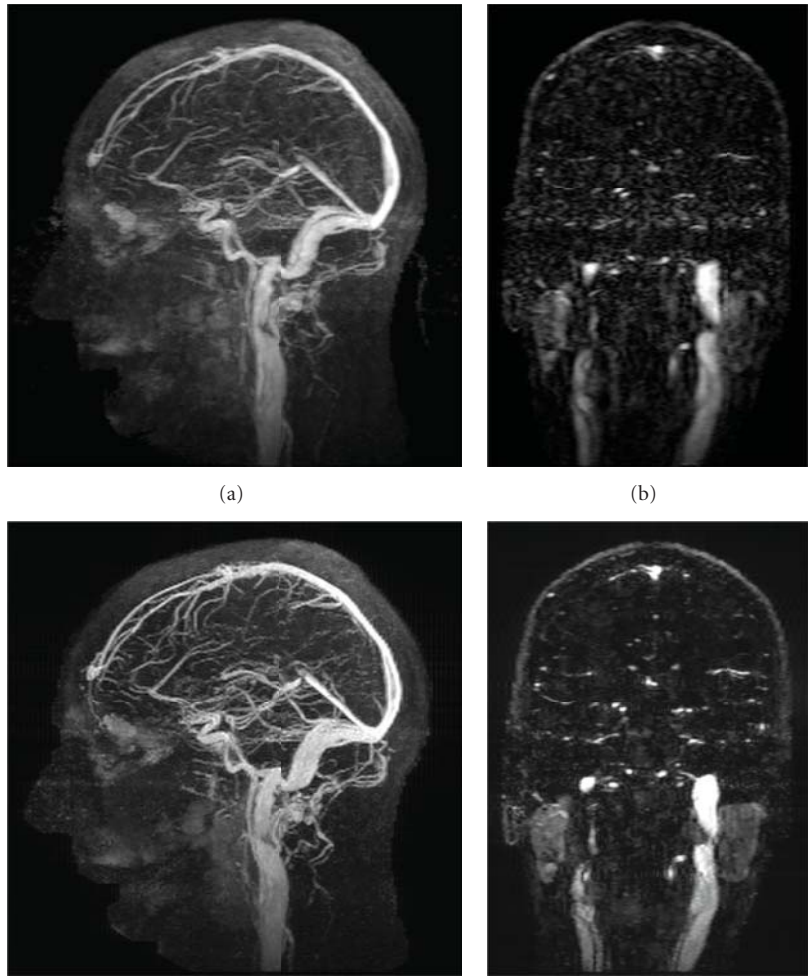


Figure 1: Sagittal maximum intensity projection (MIP) images (column 1) and coronal cross-section images (column 2) for test data set 5. (a-b) represent the current clinical reconstruction protocol result, and (c-d) represent the CS reconstruction. Note the relatively superior vascular conspicuity and parotid gland homogeneity in the CS reconstruction images.

It is an ×86-based many-core processor based on small in-order cores that combines the full programmability of today's general-purpose CPU architectures with the compute throughput and memory bandwidth capabilities of modern GPU architectures. Each core is a general-purpose processor, which has a scalar unit based on the Pentium processor design, as well as a vector unit that supports 16 32-bit float or integer operations per clock. It is equipped with two levels of cache: a low latency 32 KB L1 cache and a larger globally coherent total 8 MB L2 cache that is partitioned among the cores. It offers a peak throughput of 1.2 Tflops (single-precision). Because Intel MIC is based on ×86, it provides a natural extension to the conventional ×86 programming models. Thus, we could use similar data and thread level implementation as on Core i7 processors.

3.3. Assessment of Computational Burden. Figure 2(a) shows the overview of our CS implementation. The targeted CS reconstruction algorithm is composed of multiple iterations of 3D matrix arithmetics. We divide the reconstruction model outlined in (6) into six stages based on the loop structure (denoted as *Stage1*, *Stage2*, etc.). More specifically, *Stage1*, *Stage2*, and *Stage3* correspond with computation of the left and right terms of (7), respectively. Analogously, *Stage4*, *Stage5*, and *Stage6* correspond with computation of the left and right terms of (9), respectively. Each stage performs a series of matrix computations such as elementwise additions and 3D FFTs. The pie chart in Figure 2(b) shows the execution time breakdown of the key kernels. *FFT3D* (performed in *Stage2* and *Stage5*) is the most time-consuming and accounts for 46% of the total execution time. To achieve optimal performance, FFT requires architecture-specific optimization. Thus, we use the best FFT libraries available for each architecture. Simple elementwise matrix arithmetics (*Matrix*) are the second most time-consuming kernels. Because they stream large amount of data from/to the main memory, our optimizations focus on hiding latency and utilizing bandwidth efficiently. *Diff3D* (in *Stage1* and *Stage4*) calculates the differences from the original matrix to its shifted copy. Since the same data are used multiple

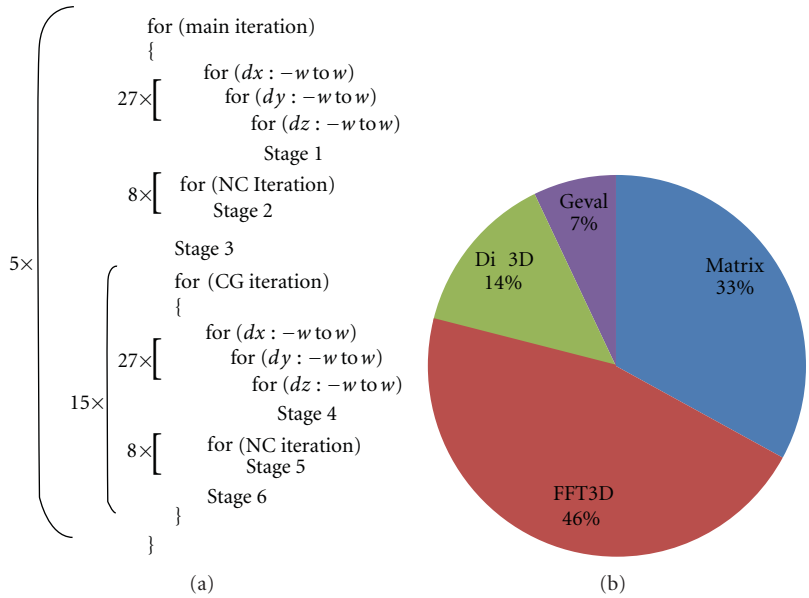


Figure 2: (a) CS implementation overview. (b) Execution time breakdown.

times, we block the matrix to exploit data reuse in fast on-die memories. *GEval* (in *Stage1* and *Stage3*) performs transcendental operations such as division and exponentiation that are implicit within (8). On GPUs, we take advantage of fast math functions; however, the performance gain due to the faster math is marginal because *GEval* comprises only 7% of the total execution time.

4. Architecture-Aware Optimization

Architecture-aware optimization can improve performance significantly. Naive implementation often misses a large amount of performance potential. In order to realize maximum performance potential, we discuss a number of important architecture-aware optimizations for our CS implementation. Our optimization techniques are general, and thus can be applied to all three architectures (CPU, GPU, and Intel MIC). In particular, since a CPU and MIC share the same programming model, an optimized CPU code can be ported to Intel MIC without much modification. However, the CUDA programming model is quite different from a CPU's. It requires significant effort to port a CPU code to GPUs, and it may be easier to program it from scratch for GPUs.

4.1. Vectorizing and Multithreading. We take advantage of modern parallel architectures through vectorizing and multithreading our CS implementation. The main data structures in CS are 3D matrices. Because each element of the matrix is computed independently in most kernels, we exploit the element-level parallelism. In other words, we can pack multiple elements into a vector computation and/or divide elements among multiple threads arbitrarily, without concern of data dependency.

We start by vectorizing the inner-most loop of 3D matrix kernels. A kernel is usually is composed of three nested loops for each of the dimensions, x, y, and z, as shown in Figure 3(a). Because there is no data dependency between elements, it is possible to vectorize within any of the iterations. However, it is most efficient to vectorize the inner-most loop, since it exhibits sequential memory accesses along the x axis. Vectorization along the y or z axis requires gathering elements from nonsequential memory locations, resulting in poor performance. In addition, most loops in CS do not contain data-dependent control flow diversions (i.e., "if" statements), which helps maintain high vector efficiency.

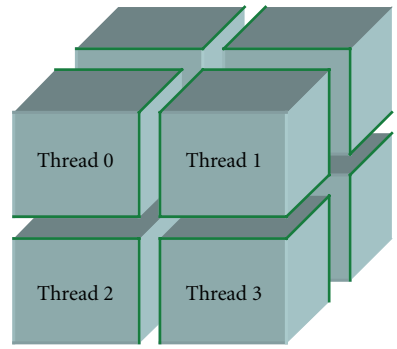
Second, we perform 3D partitioning of the 3D matrix evenly among multiple threads, as shown in Figure 3(b). For each partition, the computation requirement is almost identical, and memory access patterns are very regular. Thus, our coarse-grain static partitioning provides good load balancing. Though fine-grain dynamic partitioning may provide better load balancing, it gives rise to interthread communication overhead. In dynamic partitioning, a partition that is assigned to a thread at one stage may be assigned to another thread at another stage, which incurs data communication from the initial thread to the next. In our static partitioning, a partition is always assigned to the same thread throughout multiple stages, thus interthread communication is not required.

It is not trivial to vectorize and multithread FFT due to its butterfly access patterns and bit-reversal element shuffling. However, this has been studied for decades and optimization techniques are well known. *FFT3D* optimization in our CS implementation uses techniques discussed in [34], details of which are out of scope of this paper.

4.2. Loop Fusion. Our CS implementation performs a series of simple elementwise matrix operations. For example,

```
// z-axis
for (i = 0; i < Z; i++)
{
    // y-axis
    for (j = 0; j < Y; j++)
    {
        // x-axis
        for (k = 0; k < X; k+ = VECTOR_WIDTH)
        {
            // Vector Computations
            ...
        }
    }
}
```

(a)



(b)

Figure 3: (a) Vectoring 3-nested loop. (b) Multithreading 3D matrix.

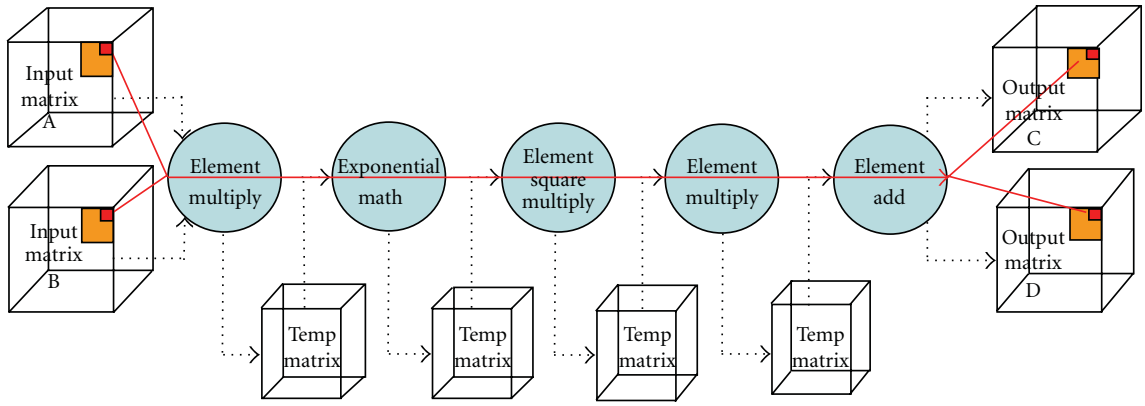


Figure 4: Example of loop fusion applied for *Stage3*.

Figure 4 shows a high-level overview of *Stage3* which corresponds to assembly of the cost functional gradient defined in (7) following construction of its elements in *Stage1* and *Stage2*. This stage is composed of five computation substages for two input matrices, A and B, and two output matrices, C and D. More specifically, A contains the set of all intermediary data generated using *Diff3D* and B contains the set of all intermediary data generated using *FFT3D*. C then corresponds to the weighting matrix defined in (8), whereas D is the entire composite variable in (7). One possible implementation is to execute each computation stage entirely before proceeding to the next state, as illustrated with the dotted arrows in the figure. For example, we first multiply matrices A and B, then we perform an exponentiation of the result, and so forth. While this approach is easy to implement, its memory behavior is inefficient. Because each stage sweeps through the entire 3D matrix and the size of the matrix is usually larger than the last level cache size, temporary matrices between stages cannot be retained within the cache, which results in cache thrashing, memory traffic increase, and, therefore, overall performance degradation. A better implementation is to block the computation so that its temporary data is kept within the cache. Main memory accesses will occur only at the beginning to read the input matrix and at the end to write the output matrix. We optimize even further to process the entire computation at the element level as shown in the solid arrow in the figure. We read an element from each of matrix A and matrix B, perform the five computations, and write the result to matrix C and D. Then, we move onto the next element, and so on. This optimization is called loop fusion, because it fuses multiple small loops of individual computations into one big loop of a combined computation. Because it handles one element at a time, data can be kept in the registers, which eliminates the need for the temporary matrix and, therefore, removes intermediate memory loads/stores completely. Also, because a fused loop performs more computation before it accesses the next element, it has more time to hide memory latency through data prefetches.

4.3. Cache Blocking through Data Partitioning. Most matrix operations in CS read/write only one element from an input matrix to an output matrix. Once an element is processed, the same element is not accessed again. However, *Diff3D* requires 27 neighbor elements to compute an output element. In other words, an input element is reused 27

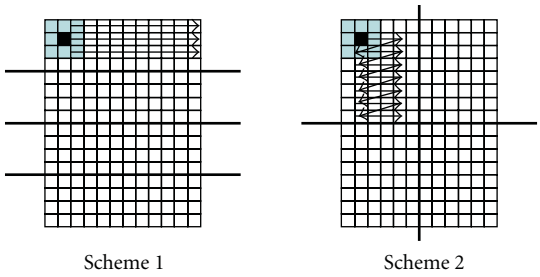


Figure 5: Cache blocking through data partitioning.

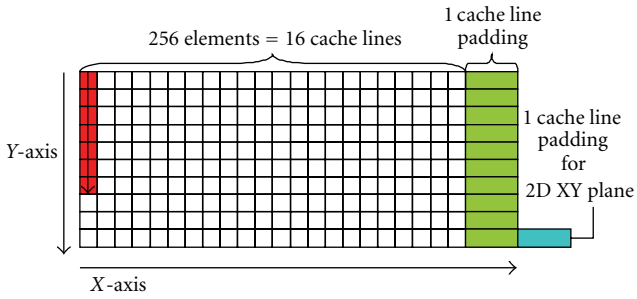


Figure 6: Cache line padding for 3D matrix.

times for 27 different output elements. To capture the data reuse, a cache-aware partitioning is required. For brevity, we explain our optimization with a 2D matrix shown in Figure 5. To compute an output, it requires 9 surrounding inputs. Scheme 1 is a cache-ignorant partitioning. It accesses elements from the beginning to the end of the current row before accessing elements in the the next row. It fails to capture data reuse if the matrix is too large. Initially, it caches the first 3×3 inputs to calculate the first output. But when it reaches the end of the row, the first 3×3 inputs are likely to be evicted from the cache if the number of elements touched during the row traversal exceeds the cache size. As a result, 3×3 inputs are reloaded from memory to calculate the first output of the next row even though 2×3 of these inputs have been already read before. Scheme 2 solves this problem by a cache-aware partitioning. It partitions the matrix in the middle of the row. Instead of moving to the end of a row, it stops at the end of the partition and moves down to the next row. It can reuse the inputs from the previous row before they are replaced from the cache. When we divide a matrix into four partitions, scheme 2 will show better cache behavior than scheme 1. We extend the same cache-aware data partitioning technique to a 3D matrix to implement *Diff3D*.

4.4. Cache Line Padding for 3D Matrix. Though the main data structure in CS is a 3D matrix, the main memory access pattern is simple streaming that accesses data from the first to the last sequentially. However, *Diff3D* and *FFT3D* also access data nonsequentially along the y and z axis. Memory accesses with a large power-of-two stride show poor cache behavior due to cache conflicts. For example, this can occur when multiple data elements from adjacent matrix rows map to the same cache line. As the result, access to the second

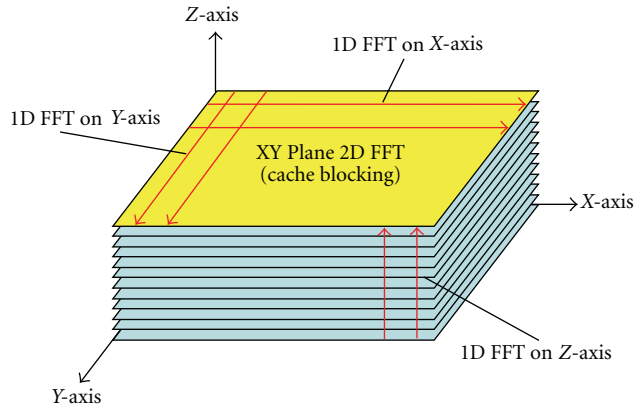


Figure 7: Last-level cache blocking for 3D FFT.

element results in the eviction of the first element from the cache. To solve the problem, we pad the matrix as shown in Figure 6. Each row along the X-axis is padded with one empty cache line at the end. Without padding, accesses along the Y-axis have stride of 16 cache lines (power of two). We break the power-of-two stride by adding one extra cache line per row, which will reduce cache conflict misses. Note that, in addition to the padding at the end of row along the X-axis, we may also need to add another padding at the end of each XY 2D plane, if the size of Y dimension is also power of two.

4.5. Last-Level Cache Blocking for 3D FFT. 3D FFT can be computed as multiple 1D FFTs along the X-, Y- and Z-axis. For a $256 \times 160 \times 80$ matrix as our reference dataset, we can first perform 12800 256-point 1D FFTs along the X-axis, followed by 20480 160-point 1D FFTs along the Y-axis, and finally 40960 80-point 1D FFTs along the Z-axis. However, performing 1D FFTs, one axis at a time is not cache efficient, because it requires sweeping the entire matrix, which incurs a lot of cache misses due to the fact that the matrix does not fit into last-level cache. Instead, we perform 2D FFTs for each 2D XY plane, then we perform 1D FFTs for the z axis, shown in Figure 7. For a given z axis value, we preload the corresponding XY plane to the cache, perform 1D FFTs along the X-axis then the y axis (a 2D FFT for the entire XY plane), and then store the resulting XY plane to the memory. Because last-level caches are usually larger than a single XY plane (320 KB for the reference dataset), our XY plane cache blocking is very effective in reducing memory bandwidth requirements. Note that larger last-level caches in Intel Core i7 processor and Intel's Knights Ferry than Tesla C2050 are beneficial for the 2D cache blocking. As each thread/core works on a different XY plane, multiple 2D blocks should be kept in the cache. In addition, as the size of datasets increases, the corresponding 2D blocks also get larger. Therefore, to achieve good system-level and dataset-level scalability, large last-level caches are critical.

4.6. Synchronization: Barrier and Reduction. CS is composed of a large number of parallel stages separated by global barriers (routines that synchronize threads in a parallel

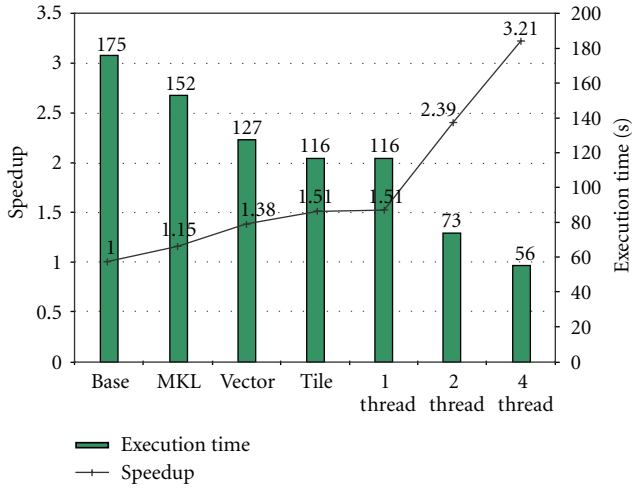


Figure 8: Impact of optimization on a quad-core CPU.

system). When threads finish computation in the parallel region, they synchronize on a barrier before proceeding to the next parallel region. Efficient barrier synchronization is paramount for high scalability. On a system with a small number of cores such as Intel Core i7 processor, barrier overhead is only ~2% of total execution time. But as we increase the number of threads (i.e., 128 threads on Intel's Knights Ferry), we observe up to ~10% overhead with our hand-optimized barrier implementation. Without the optimized barrier, the synchronization overhead would be too large, therefore resulting in poor performance. On GPUs, the barrier overhead is even worse. We implement a global barrier by launching a new kernel, that is, synchronizing with CPU, which costs one kernel launch overhead at minimum. While there exist faster barrier implementations that run entirely on GPU, they require nonstandard memory consistency model assumption.

Another synchronization primitive used in CS is reduction. In *Stage3* and *Stage6*, it reduces a 3D matrix to one scalar value. To implement this reduction, we used a software privatization technique. Each thread performs local reduction into its private scalar variable. After all threads are done, the global reduction is performed, which aggregates all local values. To implement the global reduction, we use an atomic memory operation. While atomics are generally slow on modern parallel architectures, their overhead in our CS implementation is small, due to the small fraction of time spent in the global reduction.

5. Results

We compare performance of our CS implementation on three modern parallel architectures and provide in-depth performance analysis from a computer architecture perspective.

5.1. Impact of Performance Optimization. We applied the various optimizations discussed in Section 4 to our CS implementation. We demonstrate the impact of each individual optimization on overall performance. Figure 8 shows the performance improvement of our CS implementation first as a single-threaded program on an Intel Core i7 processor as we incrementally applied our optimizations and subsequently as a multithreaded program. The vertical bar represents the execution time in seconds for each optimization step, and the line shows the corresponding relative speedup over the baseline, *Base*, which is the original single-threaded implementation of the algorithm compiled with the highest level of optimization, including autovectorization, function in-lining, and interprocedural optimization. As our first optimization, we replaced the FFTW [35] used in the original implementation with the faster Intel MKL. This results in a 1.15x speedup as represented by the second bar *MKL*. Second, we hand-vectorize the codes that can not be autovectorized by the compiler. Hand-vectorization provides an additional 1.19x speedup (*Vector*). Third, we apply cache blocking to exploit data reuse in the *Diff3D* kernel, which shows another 1.10x speedup (*Tile*). Through these three single-thread optimizations, we achieve an overall 1.51x speedup over the baseline implementation. To take advantage of multiple cores/threads, we parallelize the application. For *FFT3D*, we use the parallel implementation of the MKL library, and for the other kernels, we hand-parallelize using the OpenMP library. Parallelization achieves another 1.58x speedup on two cores over the single-core baseline and a 2.14x speedup on four cores. Overall, by combining the single-thread optimization and the multithread parallelization, we achieve a 3.21x performance improvement from the baseline implementation, which reduces the total execution time from 175 seconds (*Base*) to 56 seconds (*4 Cores*).

5.2. Performance Comparison: CPU, GPU, and Intel MIC. Figure 9 compares CS performance on three architectures: Intel dual-socket six-core Core i7 processor (Intel Xeon processor X5670 at 2.93 GHz), NVIDIA Tesla C2050 (at 1.15 GHz attached to Intel Core i7 processor 960 at 3.2 GHz), and Intel's Knights Ferry (at 1.2 GHz attached to Intel Core i7 processor 960 at 3.2 GHz). We normalize the speedups with respect to the optimized quad-core Core i7 processor (Intel Core i7 processor 975 at 3.33 GHz) implementation (56 second runtime) from the previous section and show them only for dataset 1, 2, and 3. Since their dimensions are similar, the performance results for datasets 4 and 5 are correspondingly very similar to those for dataset 3 in terms of both execution time and relative performance across different architectures. Thus, for the sake of brevity, only the results for dataset 3 are shown in Figure 9. For Tesla C2050 and Knights Ferry, we show two speedup bars: one without data transfer overhead from the CPU host and the other with the overhead. The data transfer overhead results in small performance degradation, because CS spends significant time performing computation, and can hide most of the data transfer time.

The dual-socket six-core Core i7 processor (total 12 cores) performs about 1.6x faster (35 s) than the quad-core CPU (56 s), thanks to the increased core count and memory

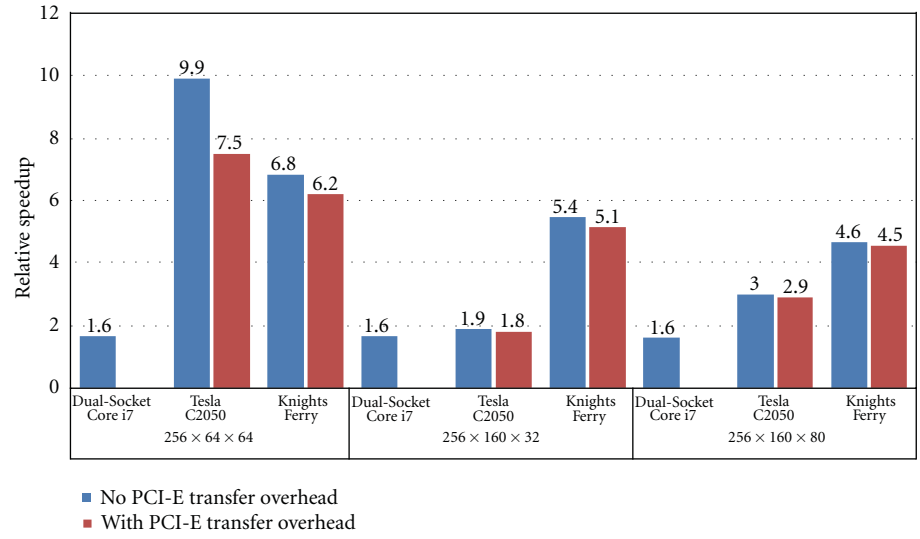


Figure 9: Performance comparison between Dual-Socket Core i7, Tesla C2050, and Knights Ferry with respect to Quad-Core Core i7.

Table 1: Performance analysis.

Kernel	Execution time breakdown	Speedup of Knights Ferry/Core i7
FFT3D	46%	~4x
Diff3D	14%	~6x
GEval	7%	~7x
Matrix	33%	~4x
Overall	100%	4.6x

bandwidth. The Tesla C2050 GPU platform is 2.9x faster than the quad-core CPU for dataset 3. However, its performance exhibits large variance across datasets. Tesla C2050 shows about 7.5x speedup for dataset 1 but shows only 1.8x speedup for dataset 2. For dataset 3 (actual clinical data), Knights Ferry achieves 4.5x speedup over the quad-core CPU, which is about 1.57x faster than Tesla C2050. Knights Ferry allows reconstructing the real anatomical dataset within a clinically feasible 12 seconds, which is a significant improvement over existing CS implementations.

Note that Core i7 processor and Knights Ferry are more efficient than Tesla C2050 in terms of resource utilization. Though Tesla C2050 has about ~4x peak flops and ~3x peak bandwidth than Core i7 processor, it only provides ~2x performance. Also, while Knights Ferry delivers ~10%± flops/bandwidth of Tesla C2050, Knights Ferry shows about 55% speedup over Tesla C2050. Finally, Tesla C2050's big performance variance across datasets is due to FFT optimization. We believe that CUFFT 3.1 is specially optimized for small power of two datasets like dataset 1. Thus, its FFT performance on dataset 1 is significantly better than on dataset 2 and 3.

6. Discussion

To elucidate the apparent impact architecture choice has on CS performance, we now provide an in-depth discussion on the performance of kernel-level operations. We then discuss the parallel scalability of CS performance to address future CMP systems.

6.1. In-Depth Performance Analysis. We provide further insights into the achieved performance by breaking down the entire application into small microkernels and analyzing the microkernels individually. We focus on the Intel quad-core Core i7 processor and Intel's Knights Ferry due to the lack of performance analysis tools in NVIDIA's CUDA environments. Table 1 shows the summary of our analysis. There are two columns for each kernel. The first column shows the fraction of execution time spent in the kernel inside the sequential code. The second column shows the speedup achieved by Knights Ferry over Core i7 processor on the kernel. Dataset 3 (single-precision complex $256 \times 160 \times 80$) is used for the analysis.

FFT3D is the most important kernel occupying 46% of the total execution time. For the reference dataset, the Intel MKL library achieves ~30 Gflops on Core i7 processor and our in-house FFT library achieves ~175 Gflops on Knights Ferry, which are the best performances that can be achieved on both architectures today. Therefore, for *FFT3D*, we estimated ~6x speedup of Knights Ferry over Core i7 processor and actually obtained ~4x speedup, which indicates that we might improve Knights Ferry performance further.

Diff3D performs a 2-point convolution. It subtracts the original matrix and its shifted matrix by dx, dy, and dz: *out = In-In_Shifted(dx, dy, dz)*. Because the computation is a simple subtraction and the data size is large (~200 MB), it is bandwidth bound. Considering the memory bandwidth of Core i7 processor and Knights Ferry, we expected ~4x speedup of Knights Ferry over Core i7 processor. In actual implementation, we achieved ~6x speedup, which indicates that Core i7 processor may have room to improve.

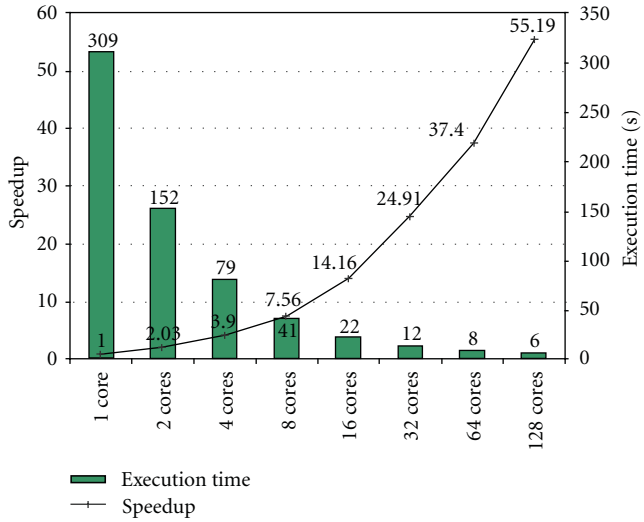


Figure 10: Performance scalability of future many-core implementations.

Table 2: Summary of performance across different hardware.

	Execution time	Speedup
Core i7 processor 1 Core, Not Optimized	175 s	1.0x
Core i7 processor 1 Core, Optimized	116 s	1.5x
Literature Best [19]	~100 s	1.7x
Core i7 processor 4 Cores	56 s	3.1x
Core i7 processor Dual-Socket 12 Cores	35 s	5.0x
NVIDIA Tesla C2050	19 s	9.2x
Intel's Knights Ferry (32 Cores)	12 s	14.5x
Research CMP Simulation (128 Cores)	6 s	29.1x

GEval involves transcendental operations. In our hand-optimized microbenchmark, exponentiation takes ~37 cycles/element in Core i7 processor and ~0.88 cycles/element in Knights Ferry, and square-root computation takes ~4.5 cycles/element in Core i7 processor and ~0.56 cycles/element in Knights Ferry. Based on the microbenchmark performance, we estimated ~8x speedup of Knights Ferry over Core i7 processor, and its actual performance (~7x) is close to our estimation.

Matrix computes simple element-wise arithmetics. In many cases, it is bandwidth bounded, because computations are simple addition, subtraction, or multiplication. However, we also optimize it by fusing multiple computations together to exploit register and cache blocking. Considering the peak memory bandwidth on both architectures, Knights Ferry is expected to be ~4x faster than Core i7 processor. Considering the peak floating point performance, Knights Ferry is expected to be ~12x faster than Core i7 processor. In reality, we achieved ~4x speedup of Knights Ferry over Core i7 processor, which indicates that the kernel is currently bandwidth-bound on both architectures.

6.2. Future CMP Scalability. Chip multiprocessors (CMPs) provide applications with an opportunity to achieve much higher performance, as the number of cores continues to grow over time in accordance with Moore's law. For CMPs to live up to their promise, it is important that as the number of cores continues to grow the performance of the application running on CMPs also increases commensurately. To gain insights into how the CS algorithm will scale on future many-core architectures, we modeled a feasible but hypothetical future CMP processor (running at 1 GHz with 256 KB/core cache) on our cycle-accurate research simulator and Figure 10 shows its CS performance scalability. We observe that CS scales well as we increase the number of cores. In particular, our 64-core configuration achieves 37x speedup over a single-core configuration, which is almost 60% parallel efficiency. Moving further, our 128-core configuration achieves 55x speedup over single-core configuration which is little less than 50% efficiency. Table 2 compiles the predicted 128-core acceleration together with the previously discussed results for Intel Core i7 processor, NVIDIA Tesla C2050, and Intel's Knights Ferry. Although a $256 \times 160 \times 80$ data volume may be considered large by numerical computing standards, it is relatively small by modern clinical standards and volumes many times larger are routinely encountered in practice. Moreover, many contemporary acquisition trends are migrating from single-phase to time-resolved paradigms, where a 3D "movie" of dynamic anatomy and physiology (e.g., of contrast flowing into and out of vessels) is created. Thus, the scalability of CMPs is paramount to seeing CS-type and other nonlinear reconstruction methods become practical for such imaging scenarios. Overall, existing and future many-core architectures are very promising platforms for accelerating the CS reconstruction algorithm to make it clinically viable.

7. Conclusion

In this work, we have shown that advanced computing architectures can facilitate significant improvements in the performance of CS MRI reconstructions and particularly that optimized use of modern many-core architectures can significantly diminish the computational barrier associated with this class of techniques. This suggests that as many-core architectures continue to evolve, CS methods can be employed in routine clinical MRI practice. Although CE-MRA was targeted in this work, the implication of the results apply to many other MRI applications as well as other areas in medical imaging such as dose reduction in computed tomography.

Acknowledgments

A preliminary version of this work was presented in [36]. The Mayo Clinic Center for Advanced Imaging Research is partially supported by the National Institute of Health (RR018898). The authors thank David Holmes III for his help in establishing the collaboration between Mayo Clinic and Intel Corporation and Stephen Riederer for providing them with access to CAPR MRI data.

References

[1] D. C. Noll, D. G. Nishimura, and A. Macovski, "Homodyne detection in magnetic resonance imaging," *IEEE Transactions on Medical Imaging*, vol. 10, no. 2, pp. 154–163, 1991.

[2] E. J. Candès, J. Romberg, and T. Tao, "Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information," *IEEE Transactions on Information Theory*, vol. 52, no. 2, pp. 489–509, 2006.

[3] D. L. Donoho, "Compressed sensing," *IEEE Transactions on Information Theory*, vol. 52, no. 4, pp. 1289–1306, 2006.

[4] M. Lustig, D. Donoho, and J. M. Pauly, "Sparse MRI: the application of compressed sensing for rapid MR imaging," *Magnetic Resonance in Medicine*, vol. 58, no. 6, pp. 1182–1195, 2007.

[5] M. Lustig, J. Santos, D. Donoho, and J. Pauly, "Rapid MR angiography with randomly under-sampled 3DFT trajectories and non-linear reconstruction," in *Proceedings of the 14th ISMRM Scientific Meeting & Exhibition*, p. 695, Seattle, Wash, USA, May 2006.

[6] T. Çukur, M. Lustig, and D. G. Nishimura, "Improving non-contrast-enhanced steady-state free precession angiography with compressed sensing," *Magnetic Resonance in Medicine*, vol. 61, no. 5, pp. 1122–1131, 2009.

[7] J. Trzasko, C. Haider, and A. Manduca, "Practical nonconvex compressive sensing reconstruction of highly-accelerated 3d parallel mr angiograms," in *Proceedings of the IEEE International Symposium on Biomedical Imaging*, pp. 274–277, July 2009.

[8] J. Trzasko, C. Haider, E. Borisch et al., "Sparse-CAPR: Highly-accelerated 4D CE-MRA with parallel imaging and nonconvex compressive sensing," *Magnetic Resonance in Medicine*. In press.

[9] A. Bilgin, T. Trouard, M. Altbach, and N. Raghunand, "Three dimensional compressed sensing for dynamic MRI," in *Proceedings of the 16th ISMRM Scientific Meeting & Exhibition*, p. 337, Ontario, Canda, May 2008.

[10] M. Doneva, H. Eggers, J. Rahmer, P. B ornert, and A. Mertins, "Highly undersampled 3d golden ratio radial imaging with iterative reconstruction," in *Proceedings of the 16th ISMRM Scientific Meeting & Exhibition*, p. 336, Ontario, Canda, May 2008.

[11] Y.-C. Kim, S. S. Narayanan, and K. S. Nayak, "Accelerated three-dimensional upper airway MRI using compressed sensing," *Magnetic Resonance in Medicine*, vol. 61, no. 6, pp. 1434–1440, 2009.

[12] C. R. Haider, H. H. Hu, N. G. Campeau, J. Huston, and S. J. Riederer, "3D high temporal and spatial resolution contrast-enhanced MR angiography of the whole brain," *Magnetic Resonance in Medicine*, vol. 60, no. 3, pp. 749–760, 2008.

[13] C.-H. Chang and J. Ji, "Compressed sensing MRI with multi-channel data using multi-core processors," in *Proceedings of the 31st Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC '09)*, pp. 2684–2687, Minneapolis, Minn, USA, September 2009.

[14] C.-H. Chang and J. Ji, "Compressed sensing MRI with multi-channel data using multicore processors," *Magnetic Resonance in Medicine*, vol. 64, no. 4, pp. 1135–1139, 2010.

[15] K. Pruessmann, M. Weiger, P. Bornert, and P. Boesinger, "Advances in sensitivity encoding with arbitrary k-space trajectories," *Magnetic Resonance in Medicine*, vol. 46, no. 4, pp. 638–651, 2001.

[16] M. A. Griswold, P. M. Jakob, R. M. Heidemann et al., "Generalized autocalibrating partially parallel acquisitions (GRAPPA)," *Magnetic Resonance in Medicine*, vol. 47, no. 6, pp. 1202–1210, 2002.

[17] M. Murphy, K. Keutzer, S. Vasanawala, and M. Lustig, "Clinically feasible reconstruction time for L1-SPIRiT parallel imaging and compressed sensing MRI," in *Proceedings of the ISMRM Scientific Meeting & Exhibition*, p. 4854, Stockholm, Sweden, May 2010.

[18] M. Lustig and J. Pauly, "SPIRiT: iterative self-consistent parallel imaging recosntruction from arbitrary k-space," *Magnetic Resonance in Medicine*, vol. 64, no. 4, pp. 457–471, 2010.

[19] J. Trzasko, C. Haider, E. Borisch, S. Riederer, and A. Manduca, "Nonconvex compressive sensing with parallel imaging for highly accelerated 4D CE-MRA," in *Proceedings of the ISMRM Scientific Meeting & Exhibition*, p. 347, Stockholm, Sweden, May 2010.

[20] E. Borisch, R. Grimm, P. Rossmann, C. Haider, and S. Riederer, "Real-time high-throughput scalable MRI reconstruction via cluster computing," in *Proceedings of the 16th ISMRM Scientific Meeting & Exhibition*, p. 1492, Ontario, Canda, May 2008.

[21] J. Trzasko and A. Manduca, "Highly undersampled magnetic resonance image reconstruction via homotopic L_0-minimization," *IEEE Transactions on Medical Imaging*, vol. 28, no. 1, Article ID 4556634, pp. 106–121, 2009.

[22] A. van den Bos, "Complex gradient and Hessian," *IEE Proceedings: Vision, Image and Signal Processing*, vol. 141, no. 6, pp. 380–382, 1994.

[23] D. H. Brandwood, "A complex gradient operator and its application in adaptive array theory," *IEE Proceedings, Part F*, vol. 130, no. 1, pp. 11–16, 1983.

[24] C. R. Vogel and M. E. Oman, "Fast, robust total variation-based reconstruction of noisy, blurred images," *IEEE Transactions on Image Processing*, vol. 7, no. 6, pp. 813–824, 1998.

[25] R. Chartrand, "Exact reconstruction of sparse signals via nonconvex minimization," *IEEE Signal Processing Letters*, vol. 14, no. 10, pp. 707–710, 2007.

[26] D. Stevenson, "IEEE standard for binary floating-point arithmetic," Tech. Rep., 1985, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=30711.

[27] Nvidia's Next Generation CUDA Compute Architecture: FERMI, 2009.

[28] TESLA C2050 and C2070 Computing Processor Board, 2010.

[29] NVIDIA, "NVIDIA CUDA Programming Guide, Version 2.3.1," 2009.

[30] CUDA CUFFT Library 3.1, 2010.

[31] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09)*, pp. 1–10, ACM, New York, NY, USA, November 2009.

[32] L. Seiler, D. Carmean, E. Sprangle et al., "Larrabee: a many-core x86 architecture for visual computing," *Proceedings of SIGGRAPH*, vol. 27, no. 3, 2008.

[33] K. Skaugen, ISC 2010 Keynote.

[34] A. C. Chow, G. C. Fossum, and D. A. Brokenshire, "A programming example: large FFT on the cell broadband engine," IBM, 2005.

[35] M. Frigo et al., "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[36] D. Kim, J. D. Trzasko, M. Smelyanskiy, C. R. Haider, A. Manduca, and P. Dubey, "High-performance 3D compressive sensing MRI reconstruction," in *Proceedings of the IEEE Engineering in Medicine and Biology Society*, pp. 3321–3324, Buenos Aires, Argentina, August 2010.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 843924, 11 pages
doi:10.1155/2011/843924

Research Article

Mapping Iterative Medical Imaging Algorithm on Cell Accelerator

Meilian Xu and Parimala Thulasiraman

Department of Computer Science, University of Manitoba, Winnipeg, MB, Canada R3T 2N2

Correspondence should be addressed to Parimala Thulasiraman, thulasir@cs.umanitoba.ca

Received 9 March 2011; Accepted 3 July 2011

Academic Editor: Khaled Z. Abd-Elmoniem

Algebraic reconstruction techniques require about half the number of projections as that of Fourier backprojection methods, which makes these methods safer in terms of required radiation dose. Algebraic reconstruction technique (ART) and its variant OS-SART (ordered subset simultaneous ART) are techniques that provide faster convergence with comparatively good image quality. However, the prohibitively long processing time of these techniques prevents their adoption in commercial CT machines. Parallel computing is one solution to this problem. With the advent of heterogeneous multicore architectures that exploit data parallel applications, medical imaging algorithms such as OS-SART can be studied to produce increased performance. In this paper, we map OS-SART on cell broadband engine (Cell BE). We effectively use the architectural features of Cell BE to provide an efficient mapping. The Cell BE consists of one powerPC processor element (PPE) and eight SIMD coprocessors known as synergetic processor elements (SPEs). The limited memory storage on each of the SPEs makes the mapping challenging. Therefore, we present optimization techniques to efficiently map the algorithm on the Cell BE for improved performance over CPU version. We compare the performance of our proposed algorithm on Cell BE to that of Sun Fire ×4600, a shared memory machine. The Cell BE is five times faster than AMD Opteron dual-core processor. The speedup of the algorithm on Cell BE increases with the increase in the number of SPEs. We also experiment with various parameters, such as number of subsets, number of processing elements, and number of DMA transfers between main memory and local memory, that impact the performance of the algorithm.

1. Introduction

Medical imaging such as X-ray computed tomography has revolutionized medicine in the past few decades. The use of X-ray computed tomography has increased rapidly since 1970 when Radon's technique for reconstructing images from a set of projections was first introduced in the medical field. In 2007, it was estimated that more than 62 million scans per year were obtained in United States and about 4 million for children [1]. The number of scanners has also increased in many countries due to the ease of using these machines. The commonly used analytic technique in CT scanners to produce diagnostic evaluation of an organ or the region of interest is Fourier back projection (FBP). This technique requires a large number of projections measured uniformly over 180° to 360° [2], inducing a large amount of radiation into the body to produce quality images. Therefore, there has been a lot of interest in developing algorithms that minimize the radiation dose without impairing image quality. One such class of algorithms [2, 3] that has been studied are iterative or algebraic algorithms.

Theoretically, iterative methods require about half of the number of projections as that of FBP method [4], which makes these methods safer in terms of required radiation dose. Compared to FBP method, iterative methods have the added advantage of producing reconstructions of better quality when data is incomplete, dynamic, or noisy. These methods solve a linear system of equations and pass through many iterations of computation to reconstruct the image. Each equation corresponds to a ray. A projection angle comprises many such rays within the same angle. For the purpose of illustration, we assume Q projection angles and M rays in total.

There are basically four steps in the iterative reconstruction algorithm: (i) forward projection, (ii) error correction, (iii) back projection, and (iv) image update. The algorithm terminates when the convergence criterion is satisfied. There are several iterative algorithms in the literature. These

algorithms all follow the four steps mentioned above but differ as to when the image updates are performed. The number of updates determines the quality of the image and also gives an upper bound on the total computation time [5]. We believe that we can make use of the tremendous amount of raw computational power available on Cell BE to provide faster performance and convergence of iterative algorithms. We assume in this paper that an iteration comprises steps (i) to (iii) followed by an image update.

Historically, the algebraic reconstruction technique (ART), a ray-based method, proposed by Gordon et al. [6], is the first representative iterative algorithm. ART iterates through the three steps (one iteration) for each ray, and then updates the image at the end of step three. Note that an image update is done for each ray which is highly time consuming. Also, this is very sequential in nature. Simultaneous iterative reconstruction technique (SIRT) [7] improves upon ART and iterates through steps (i) to (iii) for all the rays before performing an image update. This method requires many iterations for accurate results and, therefore, has a slower convergence rate. Simultaneous algebraic reconstruction technique (SART) [8] combines the good properties of ART and SIRT. The algorithm works on projections. SART passes through steps (i) to (iii) for rays within one projection, followed by an image update. This is done iteratively for each of the Q projections. Note that since the image is updated after computing the rays of each of the Q projections, the convergence rate is faster and the number of iterations compared to SIRT is reduced. Both SART and SIRT produce better-quality images than ART. However, they are computationally intensive. The convergence rate of simultaneous methods can be further accelerated through ordered-subsets (OS) technique [1, 9]. Ordered subsets method partitions the projection data into disjoint subsets and processes the subsets sequentially. For ART, each ray corresponds to one subset. Therefore, for M rays, there are M subsets. In the case of SIRT, all rays (M) correspond to one subset only. A subset in SART may correspond to all the rays in one projection angle or combine several projections of different angles into one subset. This is called OS-SART [10]. Due to the fast convergence rate of SART, in this paper, we consider parallelization of SART using the ordered-subsets (OS) technique. Though OS-SART can reduce the reconstruction time with respect to the convergence rate and produce images with high quality, it is still prohibitively time consuming due to its computation-intensive nature, especially for large images with high-resolution requirements.

One approach to increase the performance of the OS-SART algorithm is to parallelize the algorithm on modern heterogeneous multicore systems which aim to reduce the gap between the application required performance and the delivered performance [11]. The cell broadband engine (Cell BE) [12] is one such architecture. This architecture supports coarse-grained data parallel applications. In OS-SART each of the subset performs the same algorithm (same instructions) supporting data parallelism.

In this paper, we use a domain-decomposition method, which subdivides the problem into disjoint subsets. In OS-SART, each subset has to be computed iteratively. Therefore, the projection angles are further subdivided and assigned to the synergetic processor elements (SPE). Each SPE can compute independently and concurrently without any communication, reducing communication overhead. Due to the limited local store on each of the SPEs, we incorporate optimization techniques in OS-SART.

The paper is organized as follows. Section 2 lists a selection of related work. Section 3 gives a brief introduction to iterative reconstruction techniques and OS-SART algorithm. Section 4 analyzes the properties and complexities of OS-SART and introduces the rotation-based projector and back-projector used in OS-SART algorithm for this paper. Section 5 lists the highlights of Cell processor, with the Cell-based OS-SART algorithm followed in Section 6. Experiment results are given in Section 7, followed by discussions in Section 8. Section 9 will conclude this paper.

2. Related Work

Compared to parallel computing research on analytic techniques, research on iterative techniques is few. Laurent et al. [13] parallelized the block-ART and SIRT methods for 3D cone beam tomography. The authors developed a fine-grained parallel implementation, which introduced more frequent communications and degraded the performance of their algorithm. Backfrieder et al. [14] used web-based technique to parallelize maximum-likelihood expectation-maximization (ML-EM) iterative reconstruction on symmetric multiprocessor clusters. A java-applet enabled web-interface was used to submit projection data and reconstruction tasks to the cluster. Li et al. [15] parallelized four representative iterative algorithms: EM, SART and their ordered subsets (OS) versions for cone beam geometry on a Linux PC cluster. They used micro-forward-back-projection technique to improve the parallelization at the ray level during forward projection.

Gordon [16] parallelized 2D ART using a linear processor array. The author investigated both sequential ART and parallel ART algorithm on different phantom data with or without noise introduced for different number of projection views. Kole and Beekman [17] parallelized ordered subset convex algorithm on a shared memory platform and achieved almost linear speedup. Melvin et al. [18] parallelized ART on a shared memory machine and observed that the speedup of ART on shared memory architectures was limited due to the frequent image update of the ray-based ART algorithm. The parallel algorithm incurred communication and synchronization overheads and degraded the performance. Subsequent to this work, Xu and Thulasiraman [19] considered the parallelization of OS-SART on shared memory homogeneous multicore architecture. The OS-SART algorithm produces higher granularity for parallelization to reduce the above-mentioned communication and synchronization latencies. The algorithm was experimented on a CPU-based shared memory machine which provides

only few dozen nodes. Due to synchronization and communication overheads of shared memory machines, the authors were unable to produce improved performance gain. In this work, the algorithm takes advantage of Cell BE's architecture: the SPEs (coprocessors) compute fine grained independent tasks, while the PPE performs the tedious tasks of gathering and distributing data. We overlap computation and communication through mechanisms such as direct memory access available on the Cell BE to tolerate synchronization and communication overheads.

Mueller and Yagel investigated SART [20], SIRT, and OS-SIRT on an older heterogeneous multicore GPU hardware. They found that the special architecture and programming model of GPU adds extra constraints on the real-time performance of ordered subset algorithms. Xu et al. [21] recently implemented OS-SIRT and SART on the GPU architecture and claimed that SART or its subsequent OS-SART is not a suitable algorithm for implementation on GPU and does not provide increased performance gain though the convergence rate is faster than SIRT.

In this paper, we show that OS-SART algorithm is suitable for parallelization on the Cell BE and compare the results to our earlier work on homogeneous multicore architecture [19].

3. Iterative Reconstruction Techniques and OS-SART Algorithm

In this section, we start with an illustration of the iterative reconstruction technique. In Figure 1, $f(x, y)$ is an unknown image of an object and p_i is a ray of one projection at an angle θ. Many such projection data may be acquired via scanners. In this paper, we assume that 1D detector array is used to acquire projection data by impinging parallel beams onto a 2D object. The object is superimposed on a square grid of $N = n^2$ cells, assuming each cell is made up of homogeneous material having a constant attenuation coefficient value f_j in the jth cell [2]. A ray is a strip of width τ in x-y plane as shown in Figure 1. In most cases, the ray width τ is approximately equal to the cell width. A line integral along a particular strip is called *raysum*, which corresponds to the measured projection data in the direction of that ray. A projection (or view or projection view) is defined as all rays projecting to the object at the same angle.

Let p_i be the *raysum* measured for ray i as shown in Figure 1. Assume that all raysums are represented using one-dimensional array. The reconstruction problem can be formulated to solve a system of linear equations as follows:

$$\sum_{j=1}^{N} w_{ij} f_j = p_i, \quad i = 1, 2, \ldots, M, \tag{1}$$

where M is the total number of rays. w_{ij} is the weighting factor that represents the contribution of jth cell along the ith ray. The weighting factor can be calculated as (i) the fractional area of the jth cell intercepted by the ith ray or (ii) the intersection length of the ith ray by jth cell when the ray width τ is small enough to be considered as a single

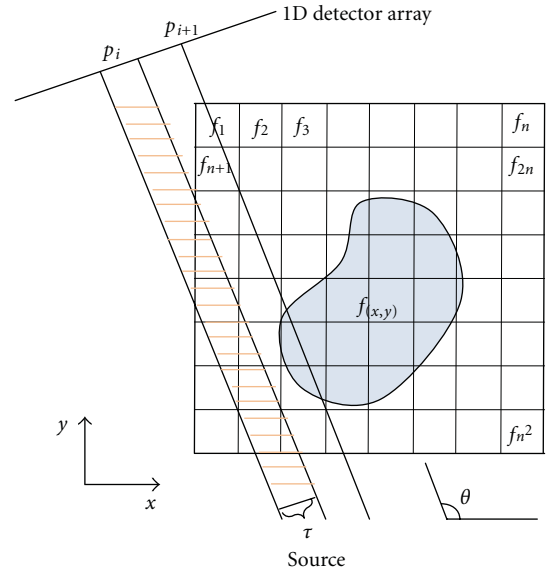


Figure 1: Illustration of iterative methods.

line. In this paper, we use the latter (Siddon's method) and will be explained in the next section. Note that for different rays, w_{ij}'s have different values for the same jth image cell. The left side of each equation in (1) is used as the forward projection operator for the specific ray i. In Figure 1, most of w_{ij}'s are zero since only a small number of cells contribute to any given *raysum*. For example, there are only ten nonzero w_{ij}'s for projection p_i if we consider using the fractional areas as the contributions.

All the rays in one projection corresponds to one subset in SART. In OS-SART, a subset may consist of many such projections. Figure 2 shows a flow chart for OS-SART. The algorithm iterates over many ordered subsets sequentially before checking the convergence criterion. The image cells are updated with

$$f_j^{r,l+1} = f_j^{r,l} + \lambda \cdot \frac{\sum_{i \in \mathrm{OS}_l} \left[\left(p_i - \sum_{k=1}^{N} w_{ik} f_k^{r,l} \right) \Big/ \sum_{k=1}^{N} w_{ik} \right] \cdot w_{ij}}{\sum_{i \in \mathrm{OS}_l} w_{ij}}, \quad j = 1, 2, \ldots N, \tag{2}$$

where p_i is the raysum of ray i, w_{ij} is the weighting factor, r is the iteration index, and l is the subset index. λ is a relaxation parameter used to reduce noise. Let, corresponding subset index (CIS), CIS $= \{1, 2, \ldots, Q\}$ correspond to indices of Q projections for the total of M rays. CIS is partitioned into T nonempty disjoint subsets OS_l, $0 \le l < T$.

Recall that each subset is computed iteratively. The computation of the pixel values for a subset, $l + 1$ requires that the subset l has already been computed and the image has been updated. Using this updated image, (2) is computed. As you can see, $f_j^{r,l+1}$ depends on the weighting factors w_{ij} and the pixel values computed for the subset l, $f_j^{r,l}$. Therefore, although there is synchronization between

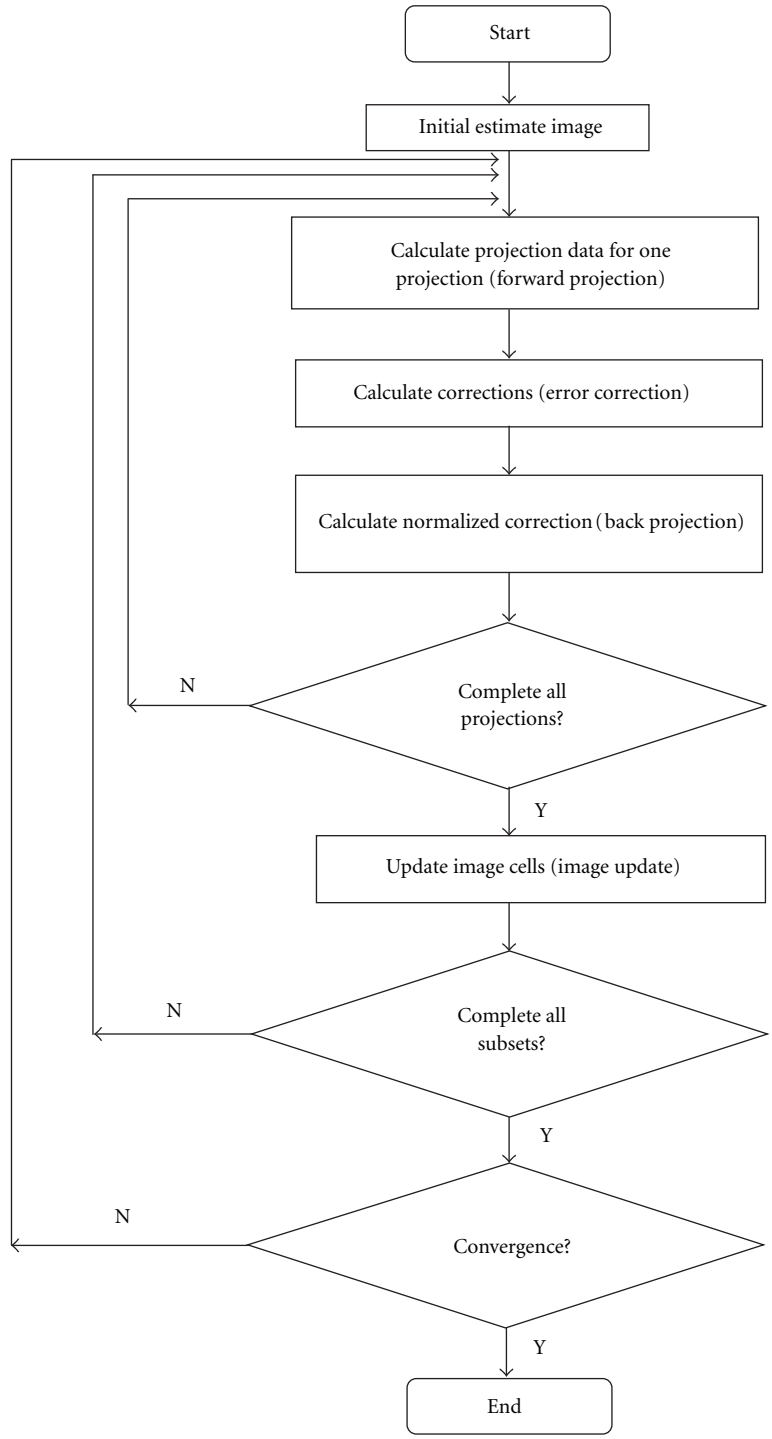


Figure 2: Framework of OS-SART reconstruction technique.

subsets, there is no synchronization within a subset. We exploit this parallelism on Cell BE.

The image estimate for each angle can be stored in main memory. The correction $((p_i - \sum_{k=1}^{N} w_{ik} f_k^{r,l}) / \sum_{k=1}^{N} w_{ik})$ and back projection $(\sum_{i \in \mathrm{OS_l}} [(p_i - \sum_{k=1}^{N} w_{ik} f_k^{r,l}) / \sum_{k=1}^{N} w_{ik}] \cdot w_{ij} / \sum_{i \in \mathrm{OS_l}} w_{ij})$ for the subset is a cumulative result of correction and back projection of different angles in the subset of the current iteration. Therefore, these can be done in parallel also. The only step that requires sequential computation in (2) is the image update. This step requires the cumulative result of the correction and back projection contributions from all angles in the subset.

The calculation of weighting factors are not only computationally intensive, they are also memory bound. In the

next section, we use a technique that saves on memory and computation for efficient computation on Cell BE.

4. Optimization Techniques on Cell BE

The sequential OS-SART algorithm is presented in Figure 2. The forward projection and back projection steps are the most time-consuming parts of the algorithm. The computation complexity of each step is: $O(I \times T \times Q/T \times n^2) = O(I \times Q \times n^2)$, where I is the total number of iterations. Let $Q = n$. Then, the computation complexity of the algorithm is $O(n^3)$, making OS-SART computationally intensive. The OS-SART algorithm is also memory bound. The memory requirement for the forward projection step includes the space required for storing the weighting factors matrix (w_{ij}) for one subset and the entire image. The space for the matrix and the image are $O(M/T \times n^2)$ and $O(n^2)$, respectively. Since M normally has the same magnitude as $N = n^2$ [2], the memory complexity of OS-SART is $O(n^4)$, making this algorithm memory intensive.

Typically, the detector array is rotated around an image, and the matrix is computed for all rays within a projection angle. For Q projections, there will exist Q such matrices. In general, the matrix w_{ij} is quite large. On the Cell BE, we are limited by the amount of memory available on each of the SPEs. Although, we could store the values in main memory, transferring data from main memory to local stores in SPE a few chunks at a time, it will degrade the performance of the algorithm due to intensive communication overhead. Therefore, in this paper, we use a rotation-based algorithm [22, 23] that is less sensitive to memory. In this method, the image is rotated around the detector array (instead of the detector array being rotated around the image) at a base angle θ. The values of w_{ij} are calculated for this angle and stored as reference. Let us call this w_{ij}^{base}. This is a one-time computation. To calculate the projection values at an angle θ_i, the forward projection starts by rotating the object at angle θ_i using bilinear interpolation method. The method then computes the forward projection data by summing over all nonzero pixels along each ray in the rotated image. That is, the pixel values are calculated using the reference matrix, w_{ij}^{base} and the rotated image. The back projection starts with the traditional back projection process, followed by rotating the object back with $-\theta_i$. Note that the main memory only stores one base weighting factor matrix which is significantly less than storing Q weighting factor matrices as in nonrotation-based methods.

As mentioned in the previous section, there are two ways of calculating the weighting factors. In this paper, we use Siddon's method [24], since it reduces the computing complexity from $O(N^3)$ of the general ray tracing method to $O(3N)$.

5. Cell Broadband Engine

The Cell BE processor is a chip multiprocessor (CMP) with nine processor elements, one PPE and eight SPEs, operating on a modified shared memory model [25]. Other important architectural features include a memory controller, an I/O controller, and an on-chip coherent bus EIB (element interconnect bus) which connects all elements on the single chip. The SPU (synergistic processing unit) in an SPE is a RISC-style processing unit with an instruction set and a microarchitecture. The PPE is the typical CPU, 64-bit PowerPC architecture which provides operating system support. The eight SPEs are purposely designed for high performance data-streaming and data-intensive computation via large number of wide uniform registers (128-entry 128-bit registers). One of the drawback of the Cell BE is the small amount of private local store (256 KB) available on each of the SPEs.

The most important difference between the PPE and SPEs is the way they access the main memory. PPE accesses the main memory directly with load and store instructions that move data between the main memory and a private register file, just like conventional processors access main memory. On the other hand, SPEs cannot access the main memory directly. They have to issue direct memory access (DMA) commands to move data and instructions between the main memory and their own local store. However, DMA transfers can be done without interrupting the SIMD operations on SPEs if the operands of SIMD operations are available in the local store. This 3-level organization of storage (register file, local store, and main memory), with asynchronous DMA transfers between local store and main memory, is a radical difference from conventional architectures and programming models [25] which complicates the programming effort by requiring explicit orchestration of data movements.

6. Cell-Based OS-SART Algorithm

There are four important routines in our proposed rotation-based OS-SART algorithm: forward projection, rotating the image, back projection, and creating reference matrix w_{ij}^{base}. By using a profiling tool, gprof, we determined the percentage of execution time spent on these routines. This was done to determine which routines require more effort in parallelization. Figure 3 shows the results for these routines for varying image sizes, with 20 subsets for 1 and 20 iterations. For both iterations, we notice that the rotation of the image is the most time consuming part. For 20 iterations, the forward projection, back projection, and rotation are also time consuming. The creation of the reference matrix is negligible. Therefore, from this figure we can see that forward projection, back projection, and rotation require efficient parallelization.

On the Cell BE, the creation of the reference matrix is computed by the PPE and stored in main memory. This is a one-time computation. The PPE controls the algorithm. It also assigns the projection angles to each of the SPEs. Given Q projection angles and T subsets, Q/T projection angles are assigned to each subset. The angles within the subset, OS_l, are further divided. For P SPEs, each SPE is assigned $Q/(T * P)$ projection angles. This process is repeated for each subset. The PPE schedules the angles to the SPEs. At the end of the

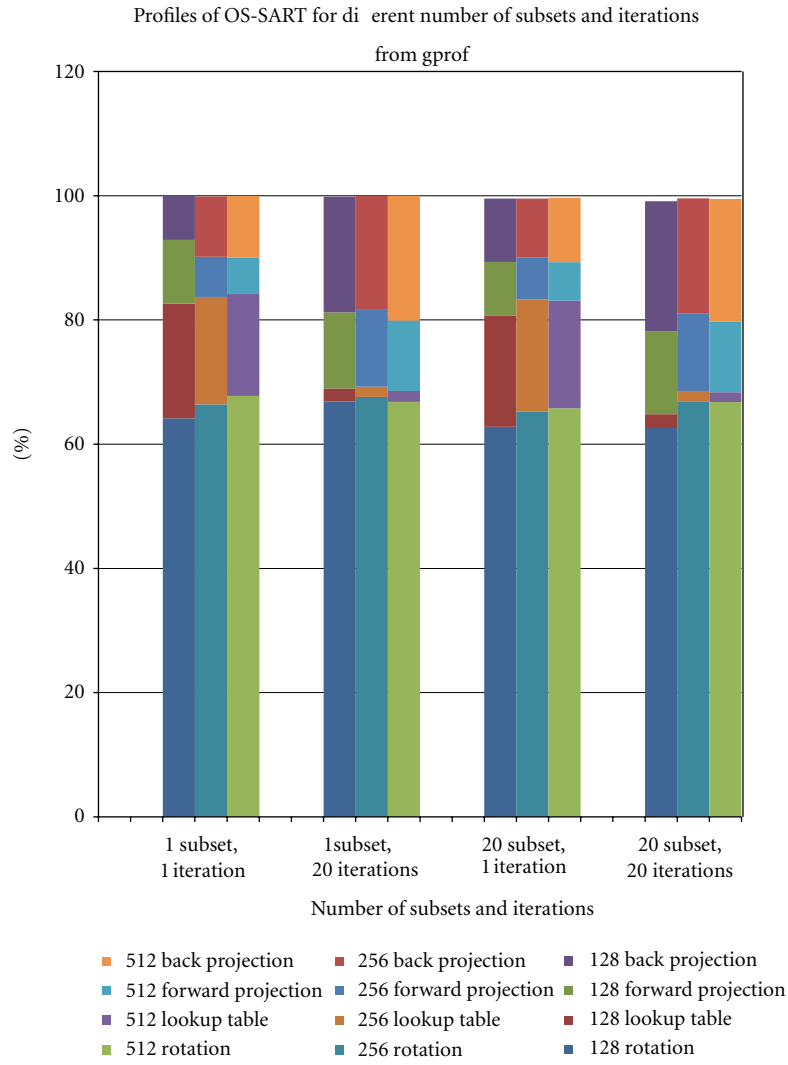


Figure 3: Profile results of OS-SART.

calculation of SPEs on a subset, the PPE performs the image update and assigns angles from the next subset, OS_{l+1}, to each SPE.

Each of the SPEs performs the following computations for their assigned angles θ_j. First, it rotates the image at an angle θ_j. Then, it computes the forward projection by accessing the reference weighting factor matrix and the image from main memory via asynchronous DMA transfers. Due to the limited local store in each of the SPEs, the matrix and image are accessed in chunks. Transferring data from main memory to local store is called DMAin [25]. Depending on the size of the image, this process may take several rounds. In the next step, the SPEs perform the error correction at the end of the forward projection computation. After error correction step, the SPE performs the back projection. The SPE sends the data back to main memory in chunks, called DMAout [25]. This is again due to the limited memory on each SPE. Finally, the SPE rotates the image back to its original position and stores this in main memory. The above process is done by an SPE for each of its assigned angles. In our paper, we balance the load on each of the SPEs by assigning the same number of projection angles.

Algorithms 1 and 2 show the pseudocode of the PPE and SPE algorithms discussed above.

7. Evaluation and Results

We have tested our proposed algorithm on two architectures: Cell BE and Sun Fire x4600. The Cell BE [26] is PowerXCell 8i processor in IBM QS22 Blade. It runs at 3.2 GHz with 16 GB of shared memory. The compiler is IBM xlc for both PPU and SPU. The Sun Fire x4600 is a distributed shared memory machine with eight AMD dual-core Opteron processors (16 cores in total) running at 1 GHz with 1 M cache per core and 4 GB memory per processor. OpenMP [27] is used for this environment.

The projection data is obtained from CTSim simulator 3.0.3 [28]. CTSim simulates the process of transmitting X-rays through phantom objects. In this work, we focus on 2D

```
Require: PPE creates threads to carry out the time-consuming parts on SPEs and setup related environments
(1) while (r < R) do
(2)     for l = 0 to T do
(3)         send messages to all SPEs to start a new subset l;
(4)         wait for all SPEs to complete the forward projection, corrections, and backprojection step;
(5)         accumulate error corrections for each pixel;
(6)         update images;
(7)     end for
(8) end while
```

Algorithm 1: Parallel OS-SART on PPE.

```
Require: receive related environment variables from PPE, p is the total number of SPEs used, Q is the total number of projections,
T is the total number of subsets,
(1) nuOfChunks = n/rowsPerDMA  {n is the one dimension size of the image, rowsPerDMA is the number of rows of the image
for each DMA transfer.}
(2) while (r < R) do
(3)     for l = 0 to T do
(4)         wait for and receive messages from PPE to start new subset l;
(5)         {go through forward projection, correction, backprojection step for assigned projections in the subset l}
(6)         for j = 0 to Q/(T × p) do
(7)             {forward projection}
(8)             locate the projection index q for the current SPE and j;
(9)             rotate the current image clockwise by the corresponding angle for projection q;
(10)            for k = 0 to nuOfChunks do
(11)                DMAin related data from the main memory, including the base weighting factors matrix, the current image;
(12)                calculate and accumulate the raysums for the forward projection step in SIMD way;
(13)            end for
(14)            {corrections}
(15)            calculate and accumulate the raysum corrections;
(16)            {bacprojection}
(17)            fork = 0 to nuOfChunks do
(18)                DMAin related data, including the weighting factors;
(19)                calculate and accumulate backprojection for each pixel in SIMD way;
(20)                DMAout the backprojection data;
(21)            end for
(22)            rotate the image counter clockwise by the corresponding angles for the projectin q;
(23)        end for
(24)    end for
(25) end while
```

Algorithm 2: Parallel OS-SART on SPE.

images to test the feasibility of our proposed parallel OS-SART algorithm. We use the Shepp-Logan phantom image of size 256×256, and 360 projections over 360 degrees. The choice of a relaxation factor has an impact on the number of iterations and reconstruction quality. The relaxation factor is usually chosen within the interval $(0.0, 1.0]$ [5]. Mueller et al. note that a smaller λ provides a less noisy reconstruction but may increase the number of iterations for convergence. The authors through their experiments conclude that λ within the interval $[0.02, 0.5]$ produces better reconstruction images with less number of iterations. Therefore, in this paper, we experiment with $\lambda = 0.2$.

Figure 4 shows the sequential computation time with varying number of subsets for both the Cell processor (1 SPE) and the AMD Opteron dual-core processor (1 core). The figure shows that the number of ordered subsets impacts the processing time for both the Cell and the Opteron processor. In both cases, execution time increases with increasing subsets. This can be easily explained as follows. As the number of subset increases, the number of image update also increases. Since the image update is done by the PPE and has to be done sequentially, the sequential portion of the algorithm, therefore, limits the performance on the entire algorithm confirming Amdhal's law. As can be seen from the speed up curve, for one subset, the algorithm running on one SPE is over 5 times faster than on one core of the AMD Opteron processor. For 360 subsets, the Cell BE is 2.7 times faster than AMD Opteron processor. Note that for larger subsets, the number of DMA transfers between the local store and main memory increases on the Cell BE,

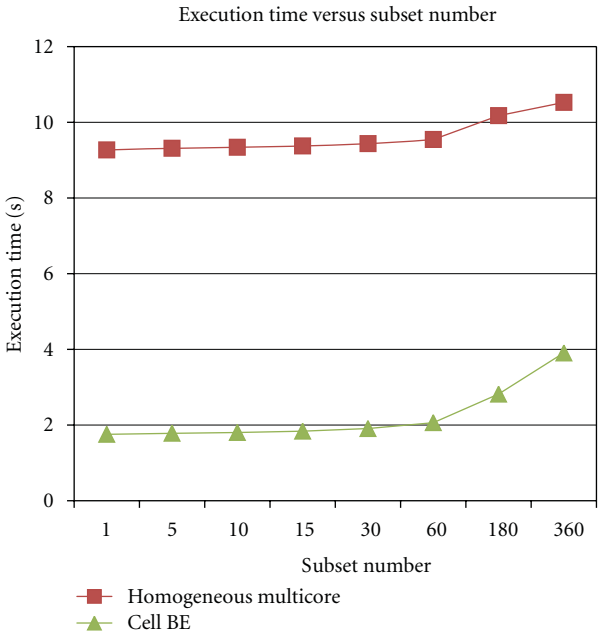


Figure 4: Computation time versus number of subsets for one iteration.

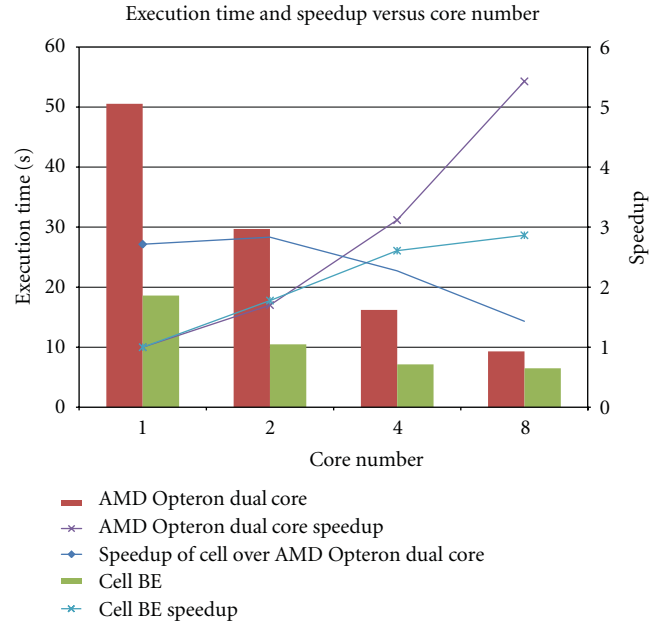


Figure 5: Computation time and speedup versus number of SPEs/cores for 20 subsets and 10 iterations.

increasing execution time. However, compared to AMD Opteron processor, the Cell BE still performs better.

Figure 5 shows the computation time and speedup for different number of SPEs and AMD cores. We set the number of subsets $T = 20$, the total number of projections, $Q = 360$, the total number of processors $P = 8$, to reconstruct the image for $I = 10$ iterations. Each subset is assigned $360/20 = 18$ projection angles. Among the 360 projection angles, we can randomly select 18 angles for each of the subsets. However, in our algorithm, we follow the equation mentioned in Section 3. That is, the ordered subset OS_l is created by grouping the projections (PR_q, $0 \le q < 360$) whose indices q satisfy $q \bmod T = l$. Therefore, for the 360 projections, OS_0 will consist of projections $0, 20, 40, \ldots, 340$. OS_1 will consist of projections $1, 21, 41, \ldots, 341$. The algorithm starts with OS_0. The 18 projection angles from OS_0 are then subdivided and assigned to SPEs. Therefore, in Figure 5, for 8 SPEs, $360/(20 * 8)$ projection angles are assigned to each SPE which performs forward projection, back projection, error correction, and rotation on their locally assigned data.

Since the Cell BE consists of 8 SPEs (processing elements or cores), our comparison on AMD Opteron is also for maximum of 8 cores. Figure 5 shows that the speedup on Cell BE is better than AMD Opteron processor when the number of processing elements used is less than 4. However, the speedup drops for Cell BE when more SPEs are used due to increased number of DMA transfers. This is due to the limited amount of local store available on each of the SPEs. As more SPEs are added, the number of DMA transfer increases since only a small amount of data can be DMAed in or DMAed out from main memory to local store and vice versa. This adds to memory latency and communication overhead. It was observed that the communication portion (including the DMA transfers and synchronization overhead) increased from 62% for one SPE to 86% for eight SPEs. The AMD HyperTransport technology attributes to the better speedup when more AMD cores are involved.

Figure 6 shows the computation and communication times of the proposed algorithm for different DMA transfer sizes. We experimented with 1, 4, 8, or 16 image rows for each DMA transfer from main memory to the local stores and vice versa. As the figure indicates, the DMA transfers significantly add to communication cost dominating the total execution time of OS-SART on Cell BE. The communication/computation ratio is significant for larger SPEs.

Figure 7 investigates the scalability of our algorithm for varying problem size and image size. As the number of SPE increases for a given problem size, the execution time decreases. The speedup of the algorithm for any image size on 8 SPEs is approximately 2.8, and the speedup increases as the number of SPE increases. Therefore, current implementation of the OS-SART with rotation-based algorithm is scalable with increasing problem and machine sizes.

Finally, Figure 8 illustrates the reconstructed images (256×256) obtained at different iterations. The number of subsets is 20. The image quality increases for more number of iterations. This result shows the accuracy of the algorithm.

8. Discussion

High-performance computing is moving towards exascale computing. Heterogeneous parallel machines with accelerators such as graphical processing units (GPUs) have demonstrated their capabilities beyond graphics rendering or general purpose computing and are proved to be well suited

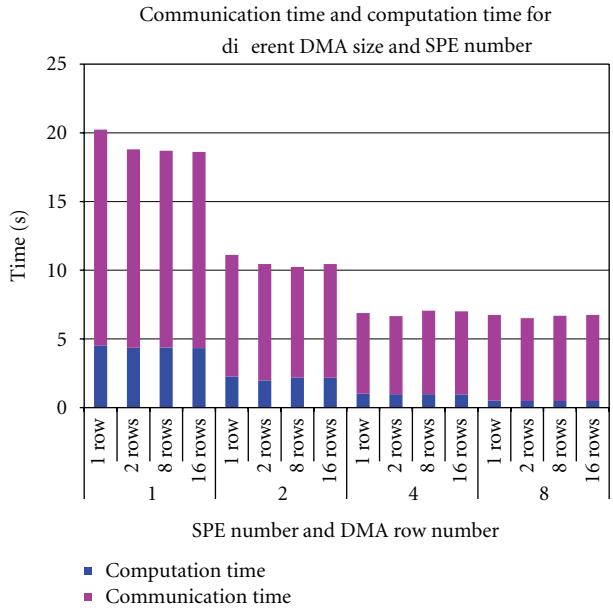


Figure 6: Computation time and communication time versus number of SPEs and number of image rows per DMA transfer for 20 subsets and 10 iterations.

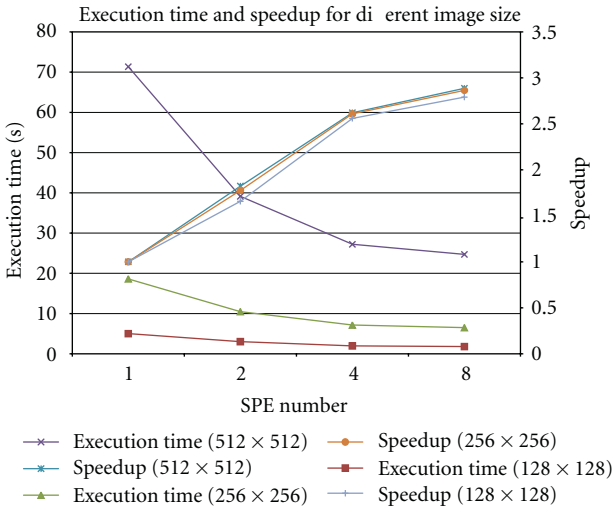


Figure 7: Computation time and speedup versus number of SPEs for different image sizes using 20 subsets and 10 iterations.

for data intensive applications. However, the communication bottleneck for data transfer between the GPU and CPU has led to the design of the AMDs accelerated processing unit (APU), which combines CPU and GPU on a single chip. This new architecture poses new challenges. First, algorithms have to be redesigned to take advantage of this architecture. In addition, the programming models differ between vendors lacking the portability of algorithms across various heterogeneous platforms. With the future of general purpose computing moving towards APUs, it is important to understand the behaviour of these architectures on high performance computing applications.

As a stepping stone to understand the applications that can be studied on APUs, we have designed, developed, and implemented the OS-SART computed tomography algorithm on on-chip accelerator, the Cell BE. Cell BE has features similar to the APU. Therefore, we strongly believe that the algorithm design would remain intact without any modifications. That is the major impact of our algorithm design. Our algorithm carefully takes into consideration the different components of the Cell BE, the PPE (or CPU), and SPE (SIMD processors) and subdivides the tasks accordingly. Fine-grained data intensive tasks are offloaded to SPEs, while tedious tasks of data distribution and gathering are performed by the PPE. On an APU, the PPE tasks can be computed by the CPU and SPE tasks by the GPUs.

Porting of the algorithms from Cell BE to AMD APU is not straight forward due to the different programming paradigm. However, recently, OpenCL has been regarded as the standard programming model for heterogeneous platforms. The parallel code used in the paper can be rewritten in OpenCL providing easy portability onto the APUs.

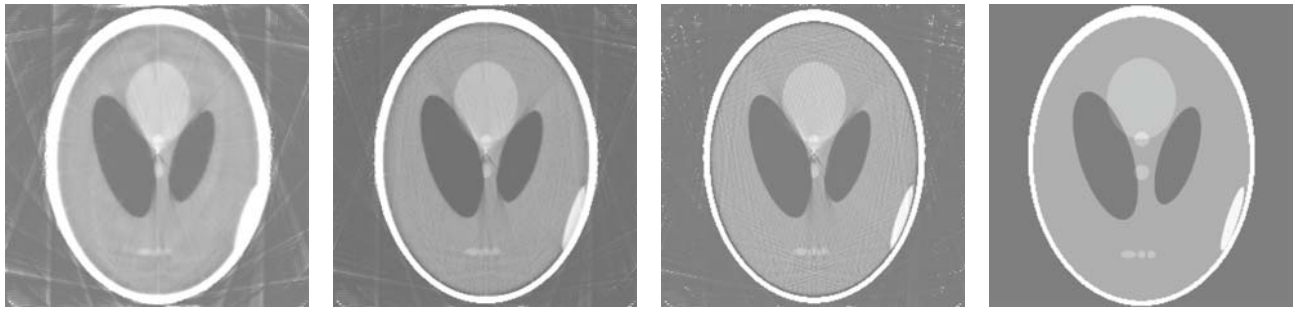
One of the drawback of Cell BE is its limited memory storage on SPEs. The APU rectifies this with its large GPU memory size. The many cores available on the GPU will allow increased number of iterations for more accuracy for the same data size used in this paper without degrading the performance. We will also have the ability to experiment with larger data sizes.

Finally, in commercial CT machines, the Fourier back projection method is the algorithm of choice. This is partly due to the tremendous amount of computational power (required by iterative techniques) only obtained through supercomputers, making them unusable or unaffordable due to very high computational cost. However, with powerful general purpose computers in the market, it should be easy to develop iterative algorithms for use in real time to help medical practitioners with real time diagnosis.

9. Conclusions and Future Work

In this paper, we efficiently mapped the OS-SART algorithm using the architectural features of the Cell BE. One of the main drawback of the Cell BE is the limited memory storage on each of the SPEs. To circumvent this problem, we used rotation-based algorithm that incorporates a technique to calculate the projection angles using less memory. Though this was efficient, it also added to the number of transfers required to DMAin and DMAout the data from main memory to local store on SPE, which was a bottleneck as the number of SPEs increased. However, in comparison to a shared memory machine, the proposed algorithm on Cell BE performed much better.

The results showed that the number of ordered subsets impacts the sequential processing time on one SPE. However, Cell-based OS-SART on one SPE was five times faster than OS-SART on AMD Opteron core for one subset and one iteration. As the number of subsets increased with number of iterations, the speedup also increased. In the future, we will modify the algorithm using double buffering to overlap



(a) iteration 5 (b) iteration 10 (c) iteration 20 (d) original shepp-Logan phantom

Figure 8: Reconstructed images at different iterations for 20 subsets.

DMA transfers with computations in order to alleviate the impact of DMA transfers.

Acknowledgments

The first author thanks Dr. Soo-Jin Lee for his description for MIPL Reconstruction Program. The second author acknowledges partial support from Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] D. J. Brenner and E. J. Hall, "Computed tomography—an increasing source of radiation exposure," *The New England Journal of Medicine*, vol. 357, no. 22, pp. 2277–2284, 2007.

[2] A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*, IEEE Press, New York, NY, USA, 1988.

[3] A. H. Andersen, "Algebraic reconstruction in CT from limited views," *IEEE Transactions on Medical Imaging*, vol. 8, no. 1, pp. 50–55, 1989.

[4] H. Guan and R. Gordon, "Computed tomography using algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography," *Physics in Medicine and Biology*, no. 41, pp. 1727–1743, 1996.

[5] K. Mueller, *Fast and accurate three dimensional reconstruction from cone-beam projecton data using algebraic methods*, Ph.D. thesis, The Ohio State University, Columbus, Ohio, USA, 1998.

[6] R. Gordon, R. Bender, and G. T. Herman, "Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography," *Journal of Theoretical Biology*, vol. 29, no. 3, pp. 471–481, 1970.

[7] P. Gilbert, "Iterative methods for the three-dimensional reconstruction of an object from projections," *Journal of Theoretical Biology*, vol. 36, no. 1, pp. 105–117, 1972.

[8] A. H. Andersen and A. C. Kak, "Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm," *Ultrasonic Imaging*, vol. 6, no. 1, pp. 81–94, 1984.

[9] H. M. Hudson and R. S. Larkin, "Accelerated image reconstruction using ordered subsets of projection data," *IEEE Transactions on Medical Imaging*, vol. 13, no. 4, pp. 601–609, 1994.

[10] G. Wang and M. Jiang, "Ordered-subset simultaneous algebraic reconstruction techniques (OS-SART)," *Journal of X-Ray Science and Technology*, vol. 12, no. 3, pp. 169–177, 2004.

[11] R. Banton., "5 critical factors to consider when choosing a processing solution for your HPC application," Technical Report, Mercury White Paper, 2008.

[12] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 589–604, 2005.

[13] C. Laurent, F. Peyrin, J. M. Chassery, and M. Amiel, "Parallel image reconstruction on MIMD computers for three-dimensional cone-beam tomography," *Parallel Computing*, vol. 24, no. 9-10, pp. 1461–1479, 1998.

[14] W. Backfrieder, S. Benkner, and G. Engelbrecht, "Web-based parallel ML EM reconstruction for SPECT on SMP clusters," in *Proceedings of the International Conference on Mathematics and Engineering Techniques in Medicine and Biological Science*, Las Vegas, Nev, USA, 2001.

[15] X. Li, J. Ni, and G. Wang, "Parallel iterative cone beam CT image reconstruction on a PC cluster," *Journal of X-Ray Science and Technology*, vol. 13, no. 2, pp. 1–10, 2005.

[16] D. Gordon, "Parallel ART for image reconstruction in CT using processor arrays," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 21, no. 5, pp. 365–380, 2006.

[17] J. S. Kole and F. J. Beekman, "Parallel statistical image reconstruction for cone-beam x-ray CT on a shared memory computation platform," *Physics in Medicine and Biology*, vol. 50, no. 6, pp. 1265–1272, 2005.

[18] C. Melvin, M. Xu, and P. Thulasiraman, "HPC for iterative image reconstruction in CT," in *Proceedings of the ACM Canadian Conference on Computer Science and Software Engineering, (C3S2E '08)*, Quebec, Canada, May 2008.

[19] M. Xu and P. Thulasiraman, "Rotation based algorithm for parallelizing OS-SART for CT on homogeneous multicore architecture," in *Proceedings of The Twelfth IASTED International Conference on Signal and Image Processing*, Maui, Hawaii, USA, Augest 2010.

[20] K. Mueller and R. Yagel, "Rapid 3-D cone-beam reconstruction with the simultaneous algebraic reconstruction technique (SART) using 2-D texture mapping hardware," *IEEE Transactions on Medical Imaging*, vol. 19, no. 12, pp. 1227–1237, 2000.

[21] F. Xu, K. Mueller, M. Jones, B. Keszthelyi, J. Sedat, and D. Agard, "On the efficiency of iterative ordered subset reconstruction algorithms for acceleration on GPUs," in *Proceedings of the 11th International Conference on Medical Image Computing and Computer Assisted Intervention, (MICCAI '08)*, New York, NY, USA, Sept 2008.

[22] E. V. R. Bella, A. B. Barclay, and R. W. Schafer, "A Comparison of rotation-based methods for iterative reconstruction algorithms," *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 3370–3376, 1996.
[23] S. Lee and S. Kim, "Performance comparison of projector-backprojector paris for iterative tomographic reconstruction," in *Proceedings of the SPIE, Applications of Digital Image Processing*, pp. 656–667, Bellingham, Wash, USA, 2003.
[24] R. L. Siddon, "Fast calculation of the exact radiological path for a three-dimensional CT array," *Medical Physics*, vol. 12, no. 2, pp. 252–255, 1985.
[25] A. Arevalo, R. M. Matinata, M. Pandian et al., "Programming the cell broadband engine examples and best practices," Technical Report, IBM Redbooks, 2007.
[26] Sharcnet, http://www.sharcnet.ca.
[27] OpenMP, http://www.openmp.org.
[28] Ctsim, http://www.ctsim.org.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 137604, 15 pages
doi:10.1155/2011/137604

Research Article

On the Usage of GPUs for Efficient Motion Estimation in Medical Image Sequences

Jeyarajan Thiyagalingam,[1,2] Daniel Goodman,[1,3] Julia A. Schnabel,[4] Anne Trefethen,[1,2] and Vicente Grau[1,4]

[1] *Oxford e-Research Centre, University of Oxford, Oxford OX1 3QG, UK*
[2] *Institute for the Future of Computing, Oxford Martin School, University of Oxford, Oxford OX1 3BD, UK*
[3] *School of Computer Science, The University of Manchester, Manchester M13 9PL, UK*
[4] *Institute of Biomedical Engineering, Department of Engineering Science, University of Oxford, Oxford OX3 7DQ, UK*

Correspondence should be addressed to Jeyarajan Thiyagalingam, jeyarajan.thiyagalingam@oerc.ox.ac.uk

Received 1 April 2011; Accepted 3 June 2011

Academic Editor: Yasser M. Kadah

Images are ubiquitous in biomedical applications from basic research to clinical practice. With the rapid increase in resolution, dimensionality of the images and the need for real-time performance in many applications, computational requirements demand proper exploitation of multicore architectures. Towards this, GPU-specific implementations of image analysis algorithms are particularly promising. In this paper, we investigate the mapping of an enhanced motion estimation algorithm to novel GPU-specific architectures, the resulting challenges and benefits therein. Using a database of three-dimensional image sequences, we show that the mapping leads to substantial performance gains, up to a factor of 60, and can provide near-real-time experience. We also show how architectural peculiarities of these devices can be best exploited in the benefit of algorithms, most specifically for addressing the challenges related to their access patterns and different memory configurations. Finally, we evaluate the performance of the algorithm on three different GPU architectures and perform a comprehensive analysis of the results.

1. Introduction

Motion estimation is one of the fundamental and crucial operations in machine vision and in video-processing applications. The process is often computationally intensive, and minimising the time for estimation across a number of frames is often a key objective in interactive image/video processing applications. As we will see in forthcoming sections, the task is repetitive and renders itself for exploitation in parallel architectures. With the rise of multicore machinery, such as many-core microprocessors and graphics processing units (GPUs), it is natural that the abundant amount of parallelism available on these systems to be exploited by mapping the algorithms on them. Among these, usage of GPUs has become increasingly common across many scientific domains.

There are several reasons for such a wide adoption of GPUs across many scientific disciplines. Modern GPUs contain hundreds of computational cores and have become available at a fraction of the cost of an equivalent conventional CPU-based system. This relative measure of performance versus price and performance versus power ratios between GPU-based architectures and CPU-based architectures further encourages the choice of GPUs.

However, the performance gains are not without significant challenges. Firstly, the identification and exploitation of any parallelism in the application is the responsibility of the developers. Often, this requires extensive remapping work rather than simple program transformations and often change in the fundamental algorithm. Secondly, the GPU programming model is not oblivious to the underlying architecture. Detailed knowledge of the architecture is fundamental for writing effective GPU-based applications.

These issues are partly overcome by different programming models, such as OpenCL [1] or CUDA [2, 3]. In practice, although these programming models simplify the

task of programming these devices, they are far from providing abstractions at the domain-specific level.

GPU implementations of some specific image processing algorithms have already been made available, including optical flow algorithms as those outlined by Marzat et al. [4]. However, in this paper, we consider a more complex, complete and enhanced version of the original optical flow algorithm. The motion estimation algorithm we use in this paper combines local and global optimisations and preserves the volume during motion estimation—a key requirement for cardiovascular medical image analysis. We then map our motion estimation algorithms on to three different GPU systems with appropriate optimizations. We have chosen the systems whose GPU architectures are representative of the time line and relevant to the important architectural aspects of GPUs. We effectively demonstrate the applicability of the algorithm using a set of three-dimensional image sequences. Using this as an evaluation phase, we discuss and highlight the relative merits and demerits of architecture-based realization of the algorithm and resulting impacts on the overall performance. To the best of our knowledge, in the context of GPUs, there is no comprehensive discussion of a motion estimation algorithm of this level in the literature. Our comprehensive analysis on the effect of architecture and programming decisions can be abstracted for different image processing applications. We believe this would be a highly valuable resource for (biomedical) image analysis researchers, and this is, thus, the fundamental aim and contribution of this paper.

The rest of this paper is organized as follows: Sections 2 and 3 serve as a background for the rest of the paper. We first discuss the GPU-based systems in Section 2, highlighting the differences to the conventional CPU-based system wherever applicable. Then, we discuss the mathematics behind motion estimation in Section 3 and concisely formulate the motion estimation algorithm. This is then followed by Section 4, where we discuss the implementation and mapping aspects in detail highlighting the architectural aspects wherever necessary. We evaluate the performance of the enhanced algorithm in Section 5 along with the presentation of our analysis. Finally, we conclude the paper in Section 6 summarizing our key findings and directions for further research.

2. Parallelism with GPUs

Exploiting graphics cards or accelerator cards for their computational capability is not a new concept. However, historically they have been exceptionally hard to program and demanded programmers to have a rather in-depth understanding of the cards, their instruction set, or familiarity with OpenGL calls and Shader languages. However, with the introduction of compute unified device architecture (CUDA) by Nvidia, this setting has improved rather significantly. The CUDA is both a programming model as well as a hardware model coupled together to provide considerably high-level utilization of the GPUs. As will be observed, it is still the case that an intimate knowledge must be maintained to leverage their potential, but it is relatively easier than Shader languages or OpenGL calls.

2.1. GPU Architecture. A compute unified device architecture- (CUDA-) enabled GPU is connected to the host system via a high-speed shared bus, such as PCI Express. We show an internal arrangement of a typical GPU in Figure 1(a). Each GPU consists of an array of streaming multiprocessors. Each streaming multiprocessor is packed with a number of scalar processing cores, named streaming processors, or simply cores. This is shown in Figure 1(b). These scalar processors are the fundamental computing units which execute CUDA threads. For example, the Nvidia Tesla C2070 GPU has 14 streaming multiprocessors and each streaming multiprocessor consists of 32 streaming processors, yielding 448 processing cores in total. The number of cores per multiprocessor or the number of multiprocessors per GPU varies from device to device. In CUDA, all threads are created and managed by the hardware. As a result, the overheads are almost negligible, and this leads to the possibility of executing a large number of threads at a time and to switch between them almost instantaneously.

Unlike multicore CPUs where the processor currently contains a relatively small number of cores each of which is capable of operating completely independently of each other, computational cores inside GPUs work in tandem and in a lock-stepped fashion.

Apart from the number of computational cores, one of the important aspects on which GPUs vary from CPUs is their memory subsystem. In GPUs, the traditional control logic dedicated to data management is used for computational cores, maximizing the space for these. This renders the data placement operations to be defined by the programmer with little or no assistance from hardware. However, to facilitate better placement strategies, GPUs are equipped with different memories. For example, in conventional CPUs, data placement is voluntarily done and in the absence of any placement, the control logic is responsible for raising the data through various different memory levels, to maintain coherency and to store them. In contrast, in the context of GPUs, the programmer is responsible for moving the data to appropriate memory. Such a liberated approach leads to considerably intricate programming model. For example, the way thread contexts are handled or how the data are moved around or the guarantee on the availability of the data prior to a computation are now left to the programmer.

Recently, GPUs have evolved rather significantly in this respect. GPUs are differentiated by their compute capability. The compute capability describes the features supported by a CUDA hardware. These features vary between devices from generation to generation in respect of maximum number of threads, support for IEEE-compliant double precision and alike. GPUs with a compute capability of less than 2.0, do not support any automatic data placement or coherency mechanisms. However, from devices with compute capability of 2.0, known as Fermi-based architectures, this has changed. Fermi-based devices contain cache memories but with the possibility of performing volunteer data management. This means that even in the presence of cache memories (see

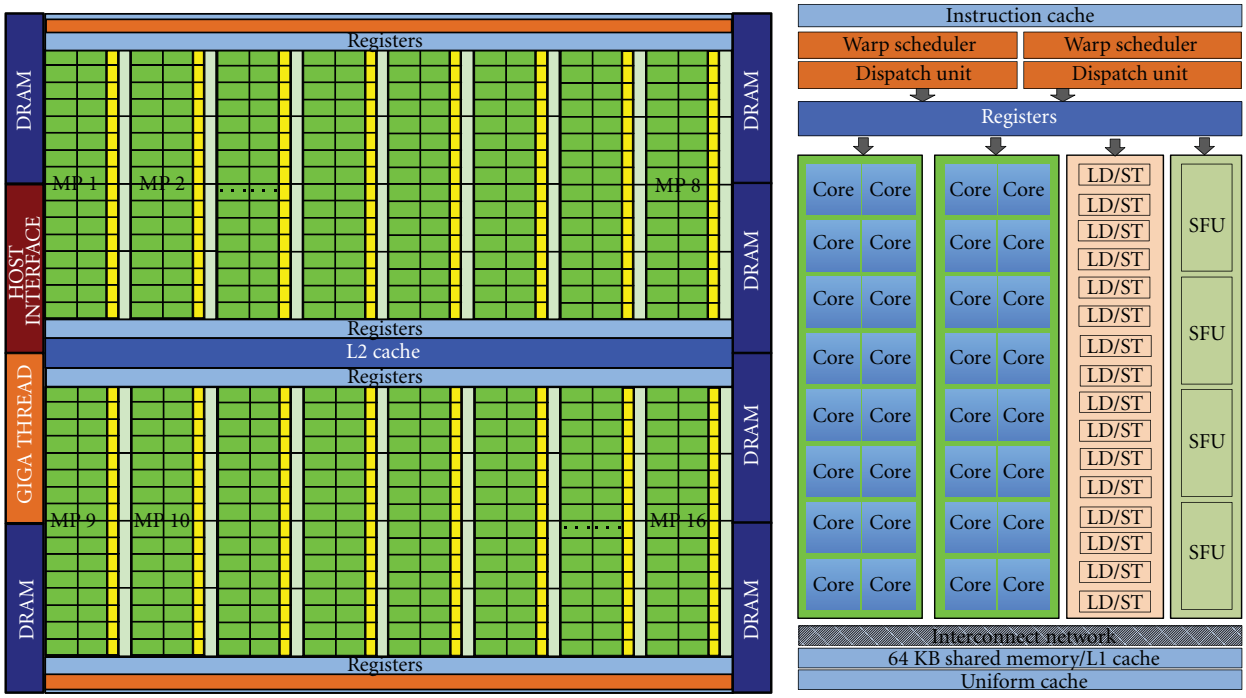


Figure 1: The overall architecture of a modern Fermi-based GPU device (a) and the inner details of a multiprocessor (b). Multiprocessors are configured around a shared level-2 cache and register files. Each multiprocessor has a number of computational cores and a level-1 cache. In the earlier versions of architectures such as C1060, these two cache levels do not exist, and the absence is facilitated by explicitly managed memories. This includes shared, constant, and texture memories. Image adopted from CUDA Programming Guide [2].

below), there is still some aspect regarding the data placement and management left to the developer. A more detailed information can be found in the CUDA Programming Guide [2].

In the latest generation of GPUs (based on Fermi architecture), the memory system is partially arranged in hierarchical manner and computational units are arranged alongside this memory system. A GPU typically has the following memory subsystems: global memory, a common level-2 cache, a combined private level-1 and shared memory, constant cache and texture cache. The global memory (also known as device memory) is common to processors (and thus to all threads) and has a high access latency. The private level-1 cache is exclusive to a streaming multiprocessor and has a low-latency connection. The level-2 cache is common across processors, and it has a better latency than global memory.

Both constant and texture memories are read-only memories separated from the shared memory. With the introduction of cache memories in GPUs, aggressive exploitation of both constant and texture memories is performed only when absolutely necessary. However, their load granularity (number of words loaded upon a load instruction) is different. As a result, sometimes, it is beneficial to utilize them. The constant memory allows each group of processors to store a selection of constants that they are going to use for the computation locally to allow fast access, without any coalesced memory access issues between processors. The texture memory provides read only access to data and follows a similar architecture to constant memory, except that instead of having a designated memory for the card as a whole, textures are bound to data stored in the card's global memory which leads to larger data capacity. As the memory is designed for storing textures in graphics applications, the memory supports a range of hardware-based functions such as interpolating the value of points that are not on integer locations.

Furthermore, the private level-1 cache is reconfigurable. This finite amount of memory pool can be configured so that part of it can be used as a shared memory, while the remaining is used as a cache memory. This enables the applications to receive partial data placement support. The system supports fixed number of such configurations, and a configuration suitable for a given application is often not known in advance and thus may need to be determined by experimentation. The total memory available for shared-memory and/or level-1 cache on current Fermi-based systems is 64 KB. This is normally used to buffer inputs and outputs to allow computations that do not naturally fit the coalesced memory access pattern to take advantage of the fast data transfers. This private first-level caches/shared memory are available to every streaming multiprocessor. In addition to this, there is a 768 KB shared secondary-level cache, which can be turned off if needed.

On GPUs, memory bandwidth to the computational cores is typically higher than that found on a CPU, meaning

that cores are less likely to suffer from starvation for data. Furthermore, this connection often has additional optimizations if certain patterns of access are adhered to and often take the form of coalesced memory accesses. If the memory is accessed by threads at random (uncoalesced), each memory load is performed independently; however, if all the cores in a group in order access consecutive memory locations (coalesced), starting from an offset into memory that is a multiple of 16, then 16 memory loads can be done in the time usually required to perform a single one.

2.2. CUDA/GPU Programming Model. The CUDA programming model, which is an extension of the C programming language, relies on this hardware support to provide concurrency. In the model, computations are expressed as special functions known as *kernels*. A kernel is launched from the host-CPU and executed by N threads using the available computational cores (and N is usually in the range of several thousands) on the GPU. All threads are organized as a one- or two-dimensional grid of thread blocks. Each block can be one-, two-, or three-dimensional. Threads in a block are assigned to the same streaming multiprocessor during execution. With a unique numbering scheme for threads, each thread can be made to compute on a different subset of the input data so that the execution leads to the single program multiple data (SIMD) style parallelism. The memory system arrangement is such that potential data locality among threads can be exploited by computational cores.

The CUDA programming model evolved over time and originally the model relied on manual placement of data—which means that the application developer is solely responsible for moving the data from the host memory to the device memory (or in reverse direction) and to exploit any reuse by relying on shared memory or constant cache. However, modern GPUs partially support automatic data placement, most specifically to cache memories. As discussed in the previous section, the level-1 cache memory can be configured as cache memory or as shared memory or as both. Yet, it is the responsibility of the programmer to make the right judgement on the amount of memory to be dedicated for cache or for shared memory and to ensure that the latencies are hidden and memory requests to the device memory are linearized for best bandwidth exploitation (hardware memory coalescing). The hardware support is available only for the data movement from and to the cache memory. In line with the conventional parallel programming models, memory transfers (corresponding to communication overheads) may offset the benefits of parallelization, if it dominates the execution time. As a result, it is performance critical that memory transfers around the system and within the GPU are minimized as much as possible. For example, if a kernel feeds another kernel with its output, it is beneficial to retain the data in the GPU device memory without any intermediate transfers to the host.

The latest generation of CUDA devices support a number of other features which we do not explore in this paper. This includes the ability to launch multiple kernels and the utilization of unified memory.

3. Motion Estimation

3.1. Background. Images are fundamental in a wide range of biomedical applications, covering most aspects of medical research and clinical practice. Improvements in technology have brought increased resolution and higher dimensionality datasets (three-dimensional and higher); furthermore, studies involving several modalities are becoming more common. In these circumstances, it is indispensable to have means for automated image analysis.

Given the dataset sizes and the limited time per patient available in clinical settings, the speed of image analysis algorithms is crucial.

Imaging technologies have become integral part of all aspects of clinical patient management, from diagnosis to guidance of minimally invasive surgical interventions. Estimation of organ motion is necessary in many of these applications, either because motion provides an indication of the presence of pathologies (as in the case of cardiac imaging), or because the presence of motion is detrimental to the accuracy of the result (as in the effect of respiratory motion in the assessment of other organs).

Motion estimation has been profusely investigated in machine vision and video coding applications, where minimising the time for estimation across a number of frames is often a key objective in interactive applications. Medical imaging shares some (but not all) aspects with these and adds the common use of three (or higher) dimensional sequences. As an example, so far widely used two-dimensional echocardiography (ultrasound imaging of the heart) is being gradually replaced by real-time 3D echocardiography (RT3D). RT3D scans can typically consist of 200^3 voxels per frame, with approximately 20 frames per scan. Local estimation of structure motion is required for assessing a range of heart conditions. In order to be fitted in the clinical protocol, this estimation would need to be done in a few seconds, ideally in real time. In Figure 2, motion estimation is illustrated in a sample echocardiographic sequence. Many other applications within the biomedical imaging field exist, in some routine clinical cases reaching image sizes of 512^3 voxels per frame, which can be much larger in basic science applications (e.g., analysis of histopathology slices).

A number of algorithms have been proposed to estimate motion in medical image sequences. In fact, the problem of motion estimation is sometimes just considered as an image registration (alignment) procedure, where registration between consecutive frames, or between each frame and a specific one selected as reference, is performed. This opens up the possibility of using any of the approaches proposed in the extensive registration literature. For an overview of registration methods, the reader is addressed to reviews such as [5–8].

In this paper, we use a motion estimation algorithm based on the optical flow approach. While we do not claim that this method is optimal for any particular task, optical flow methods are present in many state of the art algorithms for motion estimation and biomedical imaging. The particular optical flow algorithm applied here, described in Sections 3.2 and 3.3, has the additional advantage of

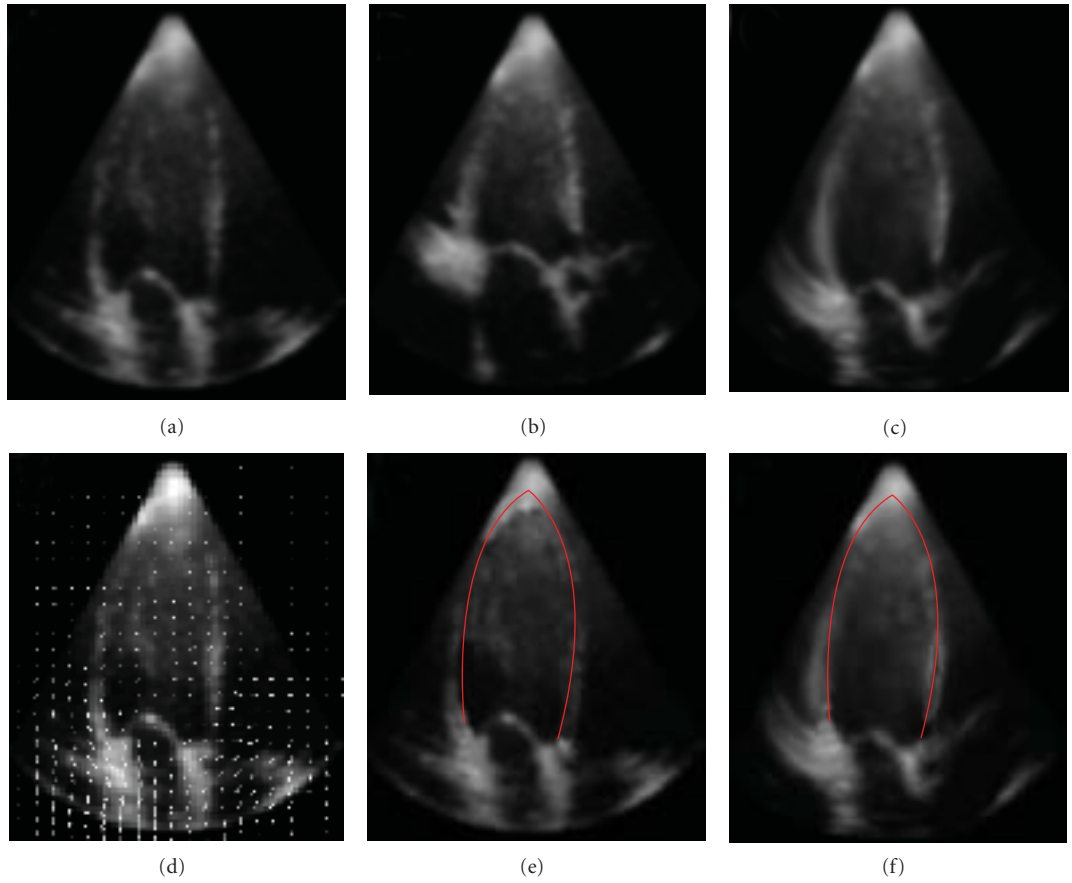


Figure 2: Example of motion estimation in a cardiac ultrasound sequence. (a) Original slice at end diastole. (b) Original slice at end systole. (c) Image (b) after alignment to image (a). (d) Estimated motion vector field, shown superimposed on (a). (e), (f) Images (a) and (c), respectively, with the endocardial contour superimposed. Note how the shapes are matched by the motion estimation process and all motion estimation and resampling were performed in 3D; a sample 2D slice is shown for clarity.

combining several generic image analysis operations (convolutions, interpolations, and iterative solution of partial differential equations). This makes it a good exemplar case to illustrate the possibilities and limitations of biomedical image analysis using GPUs, which is one of the aims of this paper.

3.2. Hybrid Motion Estimation Algorithm. In this paper, we use the motion estimation approach proposed by Bruhn et al. [9], which combines classic solutions to optical flow estimation proposed by Horn and Schunck [10] and Lucas and Kanade [11], as the baseline version. There are several reasons why we believe this is a particularly relevant algorithm for our purpose. In [9], Bruhn et al. reported excellent results including a quantitative comparison in which the algorithm is shown to outperform a number of previously published optical flow approaches. In their subsequent paper [12], they proposed different means of improving the computational performance of the algorithm in a uniprocessor platform, which makes it an excellent example to explore the peculiarities of multicore versus single-core implementations. Finally, the algorithm contains a number of individual operations which are commonly found in medical image analysis applications, and thus could be reused.

Optical flow methods are based on the assumption that corresponding points in two consecutive frames of an image sequence have the same intensity values. This condition can be linearized considering only the first terms of the Taylor expansion, which in the case of 3D images gives

$$I_x u + I_y v + I_z w + I_t = 0, \tag{1}$$

where $I_{x,y,z,t}$ are spatiotemporal partial derivatives of the image pixel intensities I and u,v, and w are displacement vector components. Equation (1) is a constraint equation and direct estimation of u, v, and w by minimising the derivatives therein is an underdetermined problem, and additional constraint(s) are required. Under this circumstance, the most that can be done is obtain the projection of the vectors in the corresponding direction of the image gradients $I_{x,y,z}$, which is referred to as the aperture problem.

Several alternatives have been proposed to solve the aperture problem. In [10], Horn and Schunck propose a variational approach, where it is assumed that the motion field is

smooth in the neighbourhood of estimation and it seeks to minimize

$$E(u,v,w) = \iint \Big(\Big(I_x u + I_y v + I_z w + I_t \Big)^2 + \alpha \Big(|\nabla u|^2 + |\nabla v|^2 + |\nabla w|^2 \Big) \Big) dx\, dy\, dz. \tag{2}$$

In other words, (1) is transformed into a cost term $(I_x u + I_y v + I_z w + I_t)^2$ to be minimized along with a regularization term $\alpha(|\nabla u|^2 + |\nabla v|^2 + |\nabla w|^2)$ which assures well-posedness. Furthermore, α is the weight of the regularization term which links intensity variation and motion.

In [11], Lucas and Kanade assume that the local motion is constant within a certain neighbourhood ρ; this provides a system of linear equations which can be directly solved. The method adopted in this paper, originally presented by Bruhn et al. [9], utilises a hybrid of both the approaches described above. This approach exploits regularization both at the local [11] and at the global level [10]. In short, the approach involves calculating the matrix

$$J_\rho(\nabla_4 I) = K_\rho * \Big(\nabla_4 I \nabla_4 I^T \Big), \tag{3}$$

where, following the notations from Bruhn et al. [9], $\nabla_4 I$ is a column vector containing the derivatives of I with respect to x, y, z, and t and K_ρ is a Gaussian kernel with variance ρ, which is convolved with each of the matrix components. J_0 is used to represent the matrix before the application of the Gaussian filter. This leads to following functional to be minimized:

$$\int_\Omega \Big(\mathbf{w}^T J_\rho(\nabla_4 I)\mathbf{w} + \alpha |\nabla \mathbf{w}|^2 \Big) dx\, dy\, dz, \tag{4}$$

along with the definitions of

$$\mathbf{w} = [u, v, w, 1]^T, \qquad |\nabla \mathbf{w}|^2 = |\nabla u|^2 + |\nabla v|^2 + |\nabla w|^2. \tag{5}$$

The functional in (4) is minimized by solving its corresponding Euler-Lagrange equations

$$\begin{aligned} 0 &= \Delta u - \frac{1}{\alpha}(J_{11}u + J_{12}v + J_{13}w + J_{14}), \\ 0 &= \Delta v - \frac{1}{\alpha}(J_{21}u + J_{22}v + J_{23}w + J_{24}), \\ 0 &= \Delta w - \frac{1}{\alpha}(J_{31}u + J_{32}v + J_{33}w + J_{34}), \end{aligned} \tag{6}$$

where Δu represents the Laplacian of u, and we use, as in [9], the notation J_{ij} to refer to the values at position (i, j) in the matrix $J_\rho(\nabla_4 I)$. With this, (6) can be expressed as a system of linear equations in the form of $Ax = b$, where

$$A = \nabla_4 I \times \nabla_4 I^T, \qquad x = \begin{bmatrix} u \\ v \\ w \end{bmatrix}, \qquad b = \begin{bmatrix} \Delta u \\ \Delta v \\ \Delta w \end{bmatrix}, \tag{7}$$

where the Laplacian for a spatial point i can be approximated from the neighbourhood elements as below:

$$\Delta u = 6u_i - \sum_{j \in N(i)} \frac{u_j}{h^2}. \tag{8}$$

We represent the three-dimensional six-neighbourhood of i, as $N(i)$ and h is the image resolution.

The total number of equations/unknowns given in (6) is $3N_xN_yN_z$, which means an iterative solving method needs to be used. In [9], Bruhn et al. used the successive over-relaxation (SOR) method. The SOR method changes the motion values on the fly; that is, the calculation of motion at iteration k will use the motion values already calculated at that iteration. An alternative is the Jacobi method, which bases the calculation of all motion values at iteration k only on the values from the previous iteration, thus allowing a more efficient parallelization. The equation to calculate u_i for each voxel i is

$$u_i^{(k+1)} = \frac{\sum_{j \in N(i)} u_j^{(k)} - (h^2/\alpha)\Big(J_{12,i} v_i^{(k)} + J_{13,i} w_i^{(k)} + J_{14,i} \Big)}{|N| + (h^2/\alpha) J_{11,i}}, \tag{9}$$

where the superscript k denotes the iteration number and $|N|$ is the number of neighbours of voxel i within the domain. $J_{12,i}$ represents the value of the component $(1, 2)$ of the J matrix in (3), calculated at voxel i in the image and calculated at the start of the iterative procedure (i.e., independent of iteration number k). Similar expressions can be easily found for the components of the motion field along the y and z axes, respectively, v and w. The algorithm thus starts with an initialization for u, v, and w (zero in our case) and a precalculation of the J values, and iteratively calculates the values of u^k, v^k, and w^k until convergence is reached. Calculation of u at voxel i, thus, requires the values of u, v, and w at the same voxel i and the values of u at neighbouring voxels j, all from the result of the previous iteration.

Using a linear approximation as in (1) works only in the case of very small motions, which is overly restrictive for general medical imaging applications. In order to overcome this limitation, it is possible to apply the whole procedure within an iterative framework, where the motion field is calculated, applied to the moving image, and the motion estimation process starts again using this newly resampled image. Note that this does not require recalculation of the spatial gradients I_x, I_y, I_z, as these are calculated on the fixed image. In the same way, the whole procedure can be embedded in a multiresolution framework without any major changes. We present the overall algorithm in Algorithm 1 and discuss in detail below.

In summary, the algorithm can be divided into these subtasks, whose GPU implementation is described below.

(1) Calculate the derivatives of the image intensities with respect to spatial and temporal coordinates: I_x, I_y, I_z, and I_t.

(2) Calculate the cross-products of the derivatives (this would correspond to the matrix $J_0(\nabla_4 I)$).

$I_1 \leftarrow Initialize()$
$I_2 \leftarrow Initialize()$
$I_x \leftarrow \mathcal{D}_x(I_1)$
$I_y \leftarrow \mathcal{D}_y(I_1)$
$I_z \leftarrow \mathcal{D}_z(I_1)$
for 1 to R
  $I_t \leftarrow \mathcal{D}_t(I_1,\ I_2)$
  $J_0(\nabla_4 I) \leftarrow \nabla_4 I \times \nabla_4 I^T$
  $J_\rho(\nabla_4 I) \leftarrow K_\rho * J_0(\nabla_4 I)$
  $A \leftarrow J_\rho(\nabla_4 I)$
  $[\Delta u, \Delta v, \Delta w] \leftarrow Jacobi\ (u,\ v,\ w,\ h,\ N_x,\ N_y,\ N_z)$
  $x \leftarrow eqnSolve(A, b)$
  $I_2 \leftarrow reSample(I_1,\ x)$
end for

Algorithm 1: The estimation algorithm without the volume preserving term (see Section 3.3. The meanings of the symbols remain as in the text. I_1 represents a static image $I(x, y, z, t)$ and I_2 represents a moving image $I(x, y, z, t + 1)$. First, the Image I_1 is initialized and partial derivatives are computed. $\mathcal{D}$ denotes the partial derivative operator. Following this, the algorithm is applied repeatedly R times. Each time of the iteration, as a precursor to solve (6), values of $\Delta u, \Delta v$ and Δw are computed using Jacobi method. The routine *Jacobi* performs this operation. One this is available, all the unknowns, given by x, are solved iteratively using the routine *eqnSolve*, which solves a system of linear equations. Then the solutions are used to resample the image to estimate the moving image I_2.

(3) Convolve each one of the components of the matrix above with a Gaussian filter K_ρ to produce $J_\rho(\nabla_4 I)$.

(4) The resulting system of linear equations given in (6) are solved using *eqnSolve* which deploys an iterative technique. This necessitates estimating the Laplacian values using *Jacobi*.

(5) Apply this motion field to all the frames of a moving image along with resampling wherever necessary.

(6) All of the above are applied repeated R times where the solution converges.

To simplify the notation, in Algorithm 1 and in subsequent sections, we assume that the motion vectors are calculated between two images I_1 and I_2, corresponding to two consecutive frames in the temporal sequence.

3.3. Enhanced Volume-Conserving Motion Estimation. Cardiac muscle is to a large extent incompressible [13, 14], and thus, in this application, it is important for the estimated motion field to preserve the original volume locally. A number of algorithms have been proposed to estimate incompressible motion fields, with the Jacobian being commonly used as a measure of volume change. In this paper, we use the variational optical flow first introduced by Song and Leahy [15], where an additional term is introduced in the minimization to favour divergence-free motion fields, which together with the diffusion-free term ensures volume preservation. Equation (4) thus becomes

$$\int_\Omega \left(\mathbf{w}^T J_\rho(\nabla_4 I)\mathbf{w} + \alpha|\nabla \mathbf{w}|^2 + \beta \cdot \mathrm{div}(\mathbf{w})\right) dx\, dy\, dz. \quad (10)$$

The solution is then computed using the Euler-Lagrange equation, similarly to the derivation presented above. The original algorithm presented in Algorithm 1 can be modified to account the preserving term we introduce here.

4. GPU Parallelization

The original algorithm exhibits abundance amount of parallelism at the pixel level. The CUDA architecture, where parallelism exists at the single instruction multiple data (SIMD) level, is particularly suitable for exploiting such a fine-grained parallelism. Although exploiting this appears rather trivial at the algorithmic level, the data placement and management posed considerable challenges in realising the algorithm. Furthermore, the continuous evolving of the architecture has a direct impact on the way the algorithms are realized.

The raw-data for the computation is represented as a vector of $N_x N_y N_z$ elements, with the best possible spatial locality along one of the dimensions (in our case, this is x). In the case where the size of the raw image to provide any undesirable effects for coalesced access, we pad the image appropriately. This leads to constant strided access along other two dimensions (stride of N_x along the y-axis and $N_x N_y$ along z-axis). Although nonlinear layouts may provide some performance benefits, we have not considered them, to minimize the addressing issues. Initial set of images will be denoted by I_1 (fixed image) and I_2 (moving image).

As we have discussed in Section 2, the complexity of modern GPU architectures in terms of data placement and management directly impacts the way that the algorithm is realized. In particular, the Fermi architecture supports both shared and cache memory. Though predetermined, a finite pool of memory can be used as a full shared memory, or as cache memory or in hybrid fashion. There is no well determined method for establishing which configuration will lead to better results. In our case, the repeated application of the algorithm may benefit from shared-memory, but this brings additional overheads to the data movement. Alternatively, the shared-memory functionality can be turned off, and we could configure the available memory as a level-1 (L1) cache, which will simplify the management. We foresee that since the accesses are constant strides, the latter configuration is likely to provide better results. However, to verify this, we implemented both methods. In the following sections, we outline how we have implemented this among a set of key functions which are central to the motion estimation algorithms outlined in Section 3.

4.1. Gradient Calculation. From the two initial three-dimensional images (I_1 and I_2), the gradient values are calculated using a forward difference approximation: $I_x(x, y, z) = I(x+1, y, z) - I(x, y, z)$, assuming that two consecutive voxels are separated by unit distance. These are calculated on the

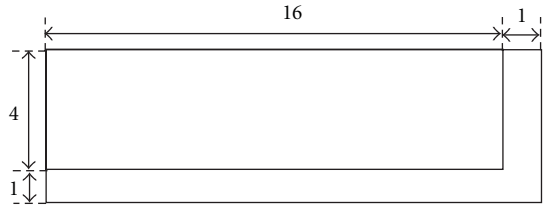


Figure 3: An example of the shared memory tiles used complete with a halo on two sides to store the extra values that are required.

fixed image I_1. In the same way, the value of the temporal derivative is approximated by the finite difference $I_2(x, y, z) - I_1(x, y, z)$. Border voxels are dealt with by assigning their corresponding derivatives to zero, rather than assigning periodic boundary conditions. The cross-products are then calculated, producing a total of nine extra values per voxel: $I_x^2, I_y^2, I_z^2 I_x I_y, I_x I_z, I_y I_z, I_x I_t, I_y I_t$, and $I_z I_t$.

The simplest strategy to parallelize this calculation would be to allocate a thread to every pixel. However, given the memory arrangement described above, this would result in each thread performing five loads from main memory (the values of I_1 at the corresponding voxel, the three forward neighbours in x, y, z and the value of I_2), including one that is uncoalesced (the value $I_1(i + 1, j, k)$ as x values are consecutive, and so they do not have an appropriate stride for coalesced memory access). To reduce this overhead, we could either rely on the L1 cache or use the shared memory, a common technique we will be reusing in realising other key routines. The data is first partitioned into cubes along the x and y directions (each cube containing the full range of z values). As the computation for each cube is looking forward, for each cube, we also require a halo of size one along the x and y directions, as shown in Figure 3. However, the partitioned cube may still not fit the shared memory. For this reason, we process the image as slices along the z direction. This introduces a halo of size one along the z direction as well. This means that the shared memory for each block of threads needs to be of size $(X + 1, Y + 1, 2)$. Image values are then loaded into shared memory, with each thread loading its interior (i.e. nonhalo) voxel in its image I_1 at that z value, and the value of the corresponding voxel in image I_2. The halo values are then loaded in by a subset of the threads. This subset is constructed by giving each thread a number k such that $k = \text{thread } Idx.x + \text{thread } Idx.y \times X$ and then selecting the threads where $k < X + Y + 1$. Each of these threads will load one value for the halo in the x and y directions. This means each thread needs to perform a maximum of three loads instead of five, with $Y + 1$ of these being uncoalesced per z value. As mentioned before, loading all the values in the z direction in one go is not possible with current shared memory sizes of only 64 KB per group of eight processors on these cards. Thus, we only store the values for z and $z + 1$ in the shared memory at any one time. However, the number of loads from global memory is kept down by moving the data around within the shared memory as the z values change. Once all the computations have been performed for the lower z plane in the image, this plane is discarded from the shared memory. Then, the $z + 1$ plane is copied allowing it to become the new z plane. Once this has been completed, a new $z + 1$ plane is loaded. Diagrams demonstrating this can be seen in Figure 4. For this application, we cut the data into pieces of size 32×4. This size was chosen through experimentation and maintaining the necessary constraint that the x dimension is a multiple of 16 in order to maintain coalesced memory accesses.

4.2. Smoothing. After the cross-products of the derivatives have been calculated, a Gaussian filter is applied to each of them. This is performed by convolving the image with a kernel approximating the Gaussian, in the following way:

$$f_\sigma(x, y, z) = \sum_{i=-K_x}^{i=K_x} \sum_{j=-K_y}^{j=K_y} \sum_{k=-K_z}^{k=K_z} g_\sigma(i, j, k) f(x - i, y - j, z - k), \tag{11}$$

with g_σ being the values of the kernel obtained by sampling a Gaussian function. The kernel has a size $(2K_x + 1, 2K_y + 1, 2K_z + 1)$, where the values of K_x, K_y, and K_z are captured as K_ρ in Algorithm 1; the values of K_x, K_y, and K_z need to be large enough to provide an accurate approximation to the Gaussian but not too large to avoid unnecessary calculations. Large values of sigma require large kernels, and thus impose a big computational load. As an alternative, the 3D convolution can be separated into three one-dimensional convolutions, one in each of the x, y, and z directions. This is the approach we have used. For simplicity of notation, in the next Section we assume $N_x = N_y = N_z$, and we use the value $K = 2N_x + 1$

As mentioned above, a naive implementation would just use a separate thread for each value within a given set. However, this approach also suffers from the number of global memory loads, which this time are a function of the kernel size (K) being applied, giving $3K$ loads from the dataset plus $3K$ accesses to load the kernel for the convolution in each direction. Of these, $3K$ accesses to load the kernel, and the K accesses to perform the convolution in the x direction will be uncoalesced. Clearly, this will cause a server bottleneck, so once again, we use the shared memory to reduce the cost of this. Additionally, we also use the constant memory on this occasion.

Half of the global memory accesses can be removed by simply storing the kernel in constant memory instead of global memory. This change does require a maximum size for the kernel to be set, but as this maximum can be in the thousands, this is not a restriction on the design.

The method for utilising the shared memory is the same as in Gradient calculation. Each thread loads some of the data ensuring coalesced memory accesses, and then all the threads share this loaded data to perform the computation. However, as the computation takes the form of three separate passes due to the separation of the three-dimensional convolution into three separate one-dimensional linear ones, it has to be formed from three separate CUDA invocations. This means that in any given invocation, the computation will only be looking in one direction. This is important, as the shared memory is not large enough to store sufficient information to

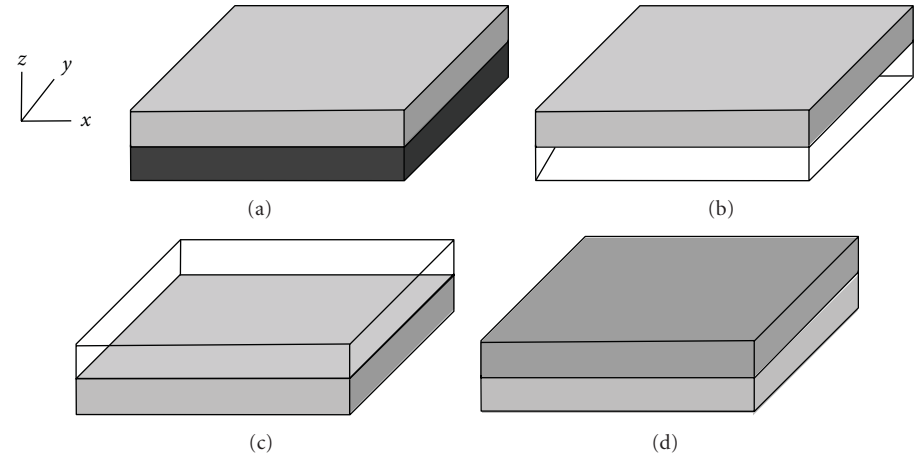


Figure 4: (a) the initial state of the shared memory. (b) after the plane z has been deleted. (c) the top tile ($z = 1$) has been moved into the position of the bottom tile (z). (d) new data with the next z coordinate value is then loaded into the space created by moving the top tile. In each instance, the different shades of Gray represent different layers within the piece of data that we are reading from into shared memory.

look in all three directions at the same time. This behaviour also means it is necessary to construct a different style of solution for the x direction to the y and z directions to take into account the need for coalesced memory accesses.

4.3. Solving the System of Linear Equations. For solving the system of linear equations, we implemented the two different variants of the Jacobi method: one version will rely on the L1-cache (without any shared memory), and the other will use the shared memory. Both approaches reduce the number of global memory accesses, in particular uncoalesced ones. In the case of the shared memory version, as before, we partition the data into tiles each with a halo, and these tiles are slices along the z dimension. However, on this occasion, given both the forward and the backward neighbours are used, the halo occupies both edges in both x and y as shown in Figure 4, and we maintain three tiles at any given moment bar the first and last values in the z direction, as zero padding at the borders is assumed. Due to the limited size of the shared memory, each of the z planes is computed in turn, so increasing the available memory for a given computation and so increasing the ratio of values computed to halo values loaded.

4.4. Intensity Interpolation. As explained above, due to the linear approximation introduced in (1) the above process has to be repeated in an iterative motion estimation/interpolation cycle. Having estimated the motion between the two images, it is now necessary to apply this motion to the moving one and interpolate the image intensities at the new positions. We use a trilinear approach, where the value of the image at each position is linearly interpolated from the values of their eight immediate neighbours. While there is a range of interpolation techniques including nearest neighbour, tricubic and different spline-based methods, in this application the trilinear method was deemed a good compromise between accuracy and speed. However, staying in line with the over arching aim of this work, we ensured that our framework is easily amenable to a different method.

The smoothness constraint in (4) and (10) means that there should be a high degree of locality associated with the reads required to sample for pixels that share locality in the original image. Despite this pattern, there is no way to determine the locations in advance, so it is not possible to use the shared memory to save on access times or to overcome the inevitable uncoalesced memory accesses. For these reasons, we turned to the texture memory to provide caching for the data accesses to improve the performance of this phase. Once the original image is mapped to a texture, it would have been possible to get the texture functionality already available in the device to perform the interpolation instead of writing new code. However, this is only implemented to a sufficient accuracy for displaying pixels on computer screens, to around four decimal places, and is inflexible in that we would be restricted to the interpolation techniques supported by the cards, rather than being able to extend this code to perform alternative interpolation techniques. As such, we map the data structure to a texture with a call from the host and then perform the interpolation on a one thread per voxel basis. Each thread calculates the location of and extracts the eight closest voxels from the texture memory. Having done this, it performs the calculation and saves the result back to the main memory. Because of the use of the texture memory as a cache, the locality of the eight pixels, and the locality of any other interpolated points within the blocks executed on a given group of cores results in both coalesced memory accesses and data reuse.

The use of the texture memory instead of the shared memory to overcome the limitations of the memory system makes this piece of code by far the simplest and demonstrates how much clearer CUDA code can be once all concerns about memory management are abstracted away. However, experiments with the use of texture memory shows that it is actually two orders slower than shared-memory counterpart, and thus, we will not be discussing this any further. We attribute the overheads to the losses.

Table 1: Details of systems used for evaluation.

Parameters	System 1	System 2	System 3
System name	C1060	GTX480	C2070
Host CPU	Xeon 5110 (Harpertown)	Intel Core i7	Xeon 5650 (Gulftown)
Host CPU speed	1.6 GHz	2.8 GHz	2.67 GHz
Host OS	Ubuntu 10.10 (64 bit)	Ubuntu 10.10 (32 bit)	Ubuntu 10.10 (64 bit)
Kernel	2.6.35	2.6.31	2.6.35
Host RAM	2 GB	4 GB	24 GB
Host L1-cache	64 KB	64 KB	64 KB
Host L2-cache	4 MB	8 MB	12 MB
GPU series	C1060	GTX480	C2070
Compute capability	1.3	2.0	2.0
Device memory	1 GB	4 GB	6 GB
Multiprocessors	24	16	14
Cores per MP	8	16	32
Total cores	192	512	498
GPU L1-cache (Shared memory)	16 KB	64 KB	64 KB
GPU L2-cache	N/A	128 KB	128 KB
CUDA version	4.0	3.3	3.2
Compiler flags	-O3 –arch = sm_13	-O3 –arch = sm_2.0	-O3 –arch = sm_2.0

5. Experimental Evaluation

5.1. Experimental Procedure. The task of performance comparison of an application on CPU- and GPU-based systems is highly dependent on a number of factors. These include the underlying operating system, compilers used, optimization flags, order of the optimizations and caching policies of the platforms in question [16]. With this in light, it is difficult to conclude that an application will always lead to performance improvement on another platform. For this reason, to gain more insight into the benefits, we use three different systems to compare and analyse the performance results. The details of the systems on which we performed our experiments are shown below in Table 1.

There are different metrics by which we could compare the benefits. In this paper, we treat the version compiled for the host CPU as the baseline version. The computation on the GPU involves more than raw-computation on the GPU cores. This includes data transfers and associated managements. However, in our context, we see that the data will persist in the GPU for subsequent runs, and therefore, we report the performance results excluding the data transfer times. For the rest of the section, we evaluated the performance of the algorithm as follows.

(1) For each system, we perform the runs a number of times in an unperturbed condition, and we chose the median of the measurements.

(2) We use a database of three-dimensional image sequences of varying sizes to test our algorithm under different configurations (see below). The database includes synthetic images wherever needed, which does not affect the results.

(3) The nonsystem-specific and algorithm-specific parameters are tested for their influence on the overall performance of the algorithm. This includes the kernel size and number of iterations.

(4) The overall motion estimation algorithm, as discussed in the previous sections, contains a number of components and each of them gain significant speedups when run on the GPU. We evaluate the speedups gained by different components.

(5) Different variants of the implementations are used to assess the impact of shared memory, L1 and L2 cache memories on the algorithm. For this, we run the following variants.

   (a) A shared memory version. This is available on all systems. In the modern systems, the L1-cache is turned off and the full pool of memory is used as shared memory. On the C1060 system, this is the standard configuration.

   (b) A nonshared memory version. On Fermi-based systems (GTX480 and C2070), this effectively turns on the L1-cache. In addition to this, this approach simplifies the overall programming as complicated techniques such as tiling are not needed, and thus purely relying on the loading resolution of the cache controllers on the GPU.

   (c) A no-L1 mode. This turns off the L1 and shared memory mode and thus purely relies on L2-cache. This configuration does not exist on the older systems (C1060).

5.2. Experimental Results. We first present the impact of the number of iterations (denoted by R in Algorithm 1) and of the kernel size (denoted by K_ρ in Alogrithm 1), on the overall speedup in Figure 5. We evaluate the impact of the number of iterations using two different fixed size images (128^3 and 256^3 images) on all three platforms, for a range of iterations. For each platform, we pick the best performing versions (among shared memory, nonshared memory and non-L1-mode) and then we vary the number of iterations. As observed, the number of iterations does not have a noticeable impact on the overall speedups across all platforms. Although varying the image size changes the maximum speedup (speedup increases as image size increases), for a given image, the number of iterations do not alter the speedup by a large degree. This is because, although increasing the number of iterations benefits from overall reuse per transfer, the inter- and intraiteration spatial locality in the cache is not in favour of the application. This essentially carries an important message: although the execution time for increased number of iterations will rise, the speed up will not be affected. However, the kernel size has an impact on

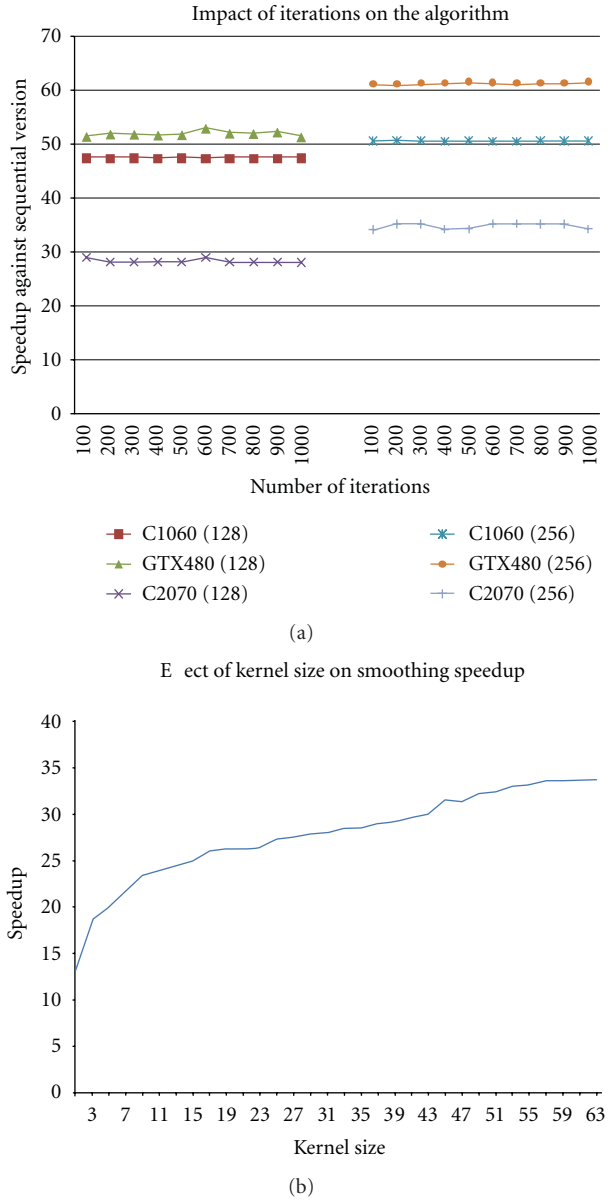


FIGURE 5: (a) shows the impact of number of iterations in the overall speed up of the algorithm. We show the impact for two different image sizes. (b) shows the impact of kernel size on the overall speed up of the algorithm.

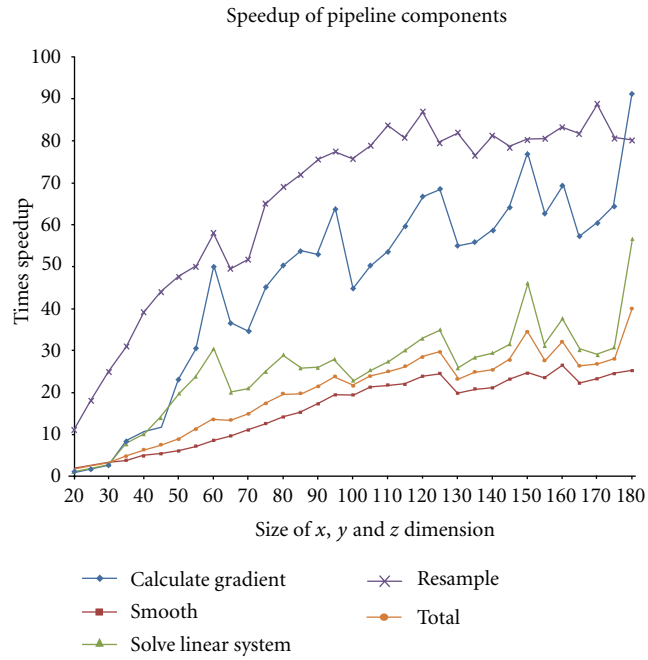


FIGURE 6: Speed up of the different components of the motion estimation algorithm for different size of images on the C1060 platform.

the overall performance of the algorithm. As the kernel size increases, the number of accesses to memory per floating point operation (FLOP) decreases and this improves the GPU speedups. Furthermore, if the kernel size were to exceed the size of the CPU cache, this observation will change considerably.

Provided that for a given case, the number of iterations and the kernel sizes are fixed, the overall speedup is only affected by the size of the image. In the remaining part of this section, we keep the number of iterations and the kernel size constant, and we only vary the image size.

As stated above, the overall motion estimation algorithm has a number of operations, which are componentized in our case, and their speedups on the GPU vary with the problem size. This is shown in Figure 6 for the C1060 platform. Here, all components of the algorithm show significant gains in speedup. The kernel size for the smoothing is 15. Different parts of the pipeline have different access patterns, leading to different overall behaviour. However, both the gradient calculation and linear system solution use the same access patterns both on the host and the GPU, explaining the similar shape. The presence of local *spikes* in speed up is due to the different optimal sizes between CPU and GPU. In our case, we observe that the Gaussian filter achieved a speed up of around 25 times, and the data generation and image interpolation achieved speedups close to 90 times in the best case. The worst case values (slowdowns) were observed for very small image sizes (e.g., around 5 times slowdown for Gaussian), which are not shown here. However, the overall speed up is much less than the best speedups of all components and this is currently limited to 60 times. We also observed the highest percentage of the time being spent solving the linear equations and smoothing the data.

Having presented the performance of fine-grained aspects of the GPU performance, we present the overall performance behaviour of the algorithm on different systems. As stated in the previous section, different variants of the implementations are tested for their performance.

The speedup on the C1060 system is shown in Figure 7 for two different configurations. One with the shared memory being exploited and the other one without any shared memory utilization. In overall, the shared memory implementation is faster than nonshared memory implementation. For the reasons discussed in Section 2, accessing shared memory is faster than accessing the device memory.

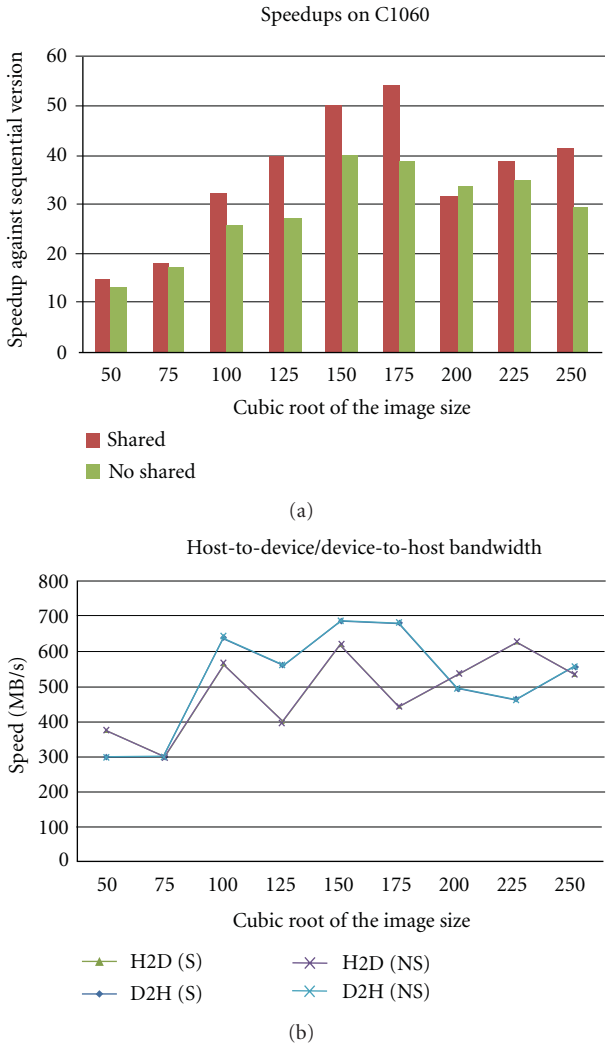


Figure 7: (a) shows the overall speed up of the algorithm on the C2070 system with the data transfer times excluded while (b) shows the variation of transfer speed with the image size.

Table 2: Raw runtimes of the algorithm on the C1060 system.

Cubic root of image size	CPU time (ms)	GPU time (with shared memory) (ms)	GPU time (without shared memory) (ms)
50	1272	86	96
75	4457	245	257
100	11362	355	443
125	21112	531	771
150	45101	895	1123
175	62310	1147	1594
200	96703	3036	2856
225	134668	3473	3834
250	176238	4261	6016

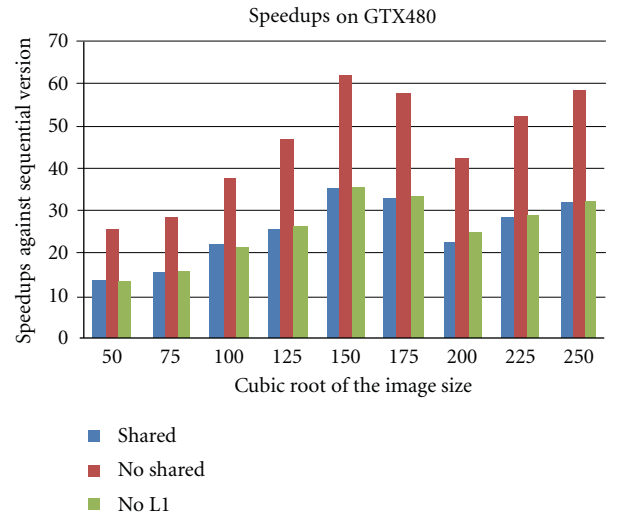


Figure 8: The overall speedup of the algorithm on the GTX480 system for different image sizes with the image size (with the data transfer times excluded).

This improved the implementation so less time is spent on wait states, thus the overall speedup gains. One other observation, for which further investigations are needed, is sudden slowdowns at image size of 200^3 which are more pronounced on the shared memory implementation.

Reporting speedups as in the figures above can be sometimes slightly misleading. In particular, this is true when slowdowns on the CPU side are translated as speedups. For this reason, we present the raw runtimes in Table 2. In addition to this, the transfer speed rate between the host and device and vice versa varies with data size. As per the configuration, the rate is biased towards the direction in which the transfer occurs (data transfer rate from device to host is much higher than transfer rates from host to device) but rarely achieves the peak rate of respective systems.

Figure 8 shows the performance of the algorithm on the GTX480 system. Since GTX480 GPU is based on the Fermi architecture, the GPU contains two levels of caches, (L1 and L2) one which can be configured either as a shared memory, as a L1-cache or a mix of both. We ran our implementation under three different configurations. First, we ran the algorithm with no-shared memory option. This is the default and triggers the usage of L1 and L2 caches of the system. Under this setup, the implementation does not need to have any extensive shared-memory mapping procedures. Then, we performed the same experiment with the shared memory variant of the implementation. Under this configuration, the L1-cache is fully configured as a shared memory system and our implementation takes the full responsibility of the loading and data placement. Finally, we run the implementation with the L1-cache being completely turned off. This renders both the shared memory and L1 not available to the application, thus fully relying on the L2 cache, which cannot be turned off.

As can be observed in Figure 8, the default configuration outperforms the other two versions. In other words, extensively tuning the application for shared memory usage is not optimal under the new architecture. Two reasons can be

Table 3: Raw runtimes of the algorithm on the GTX480 system.

Cubic root of image size	CPU time (ms)	GPU time (with shared memory) (ms)	GPU time (without shared memory) (ms)	GPU time (without L1) (ms)
50	671	49	26	49
75	2379	155	83	155
100	5788	272	150	265
125	11338	440	240	435
150	23714	677	373	671
175	31369	955	554	942
200	47914	2037	1101	1999
225	66870	2357	1275	2312
250	91610	2864	1564	2823

attributed to this failure. Firstly, the data management and placement overheads cannot compete with the automatic placement provided by the L1-cache. Secondly, the L2-cache deploys the inclusive policy, which feeds the L1 without any extra wait states.

However, turning off the L1-cache does not affect the performance significantly since the L2-cache is active. Very close inspection of Figure 8 and runtimes in Table 3 reveal that even after turning off the L1-cache, the overall performance of the shared memory implementation is outperformed. Again, we explain this based on the efficiency and loading policies of the cache controllers on GPU. Furthermore, a performance drop at the image size 175^3 is observed similar to what was observed on the C1060 system, but here, the nonshared memory implementation suffers from slowdowns.

Finally, we present the performance of the algorithm on the C2070 system in Figure 9. Since the system has no device memory, we were able to run the algorithm with considerably large image sizes (up to 525^3).

Similar to the GTX480 system, the default configuration, where no shared memory was utilized, performs better than the other two version. All the other observations align with the previous observations on GTX480 system. Furthermore, as before, even relying on the Level-2 cache is sufficient enough to gain substantial speedups. Furthermore, a performance drop observed on the C1060 and GTX480 systems observed here as well, where the observation is repeated. Although all of these observations have been visible in other systems, one subtle feature of this C2070 system is that it has a built-in error checking and correction (ECC) mechanism for the GPU memory.

Historically, in the context of GPUs, error rates were not an issue—as this only affected the color of the pixel. However, this is no longer the case with GPUs being used as part of the HPC systems as the soft error rates in DRAM are high. Despite this, previous generations of GPUs did not support ECC and only recently this support has started to arrive. In our case, C2070 supports ECC. However, the ECC comes with the penalty on runtimes due to overheads

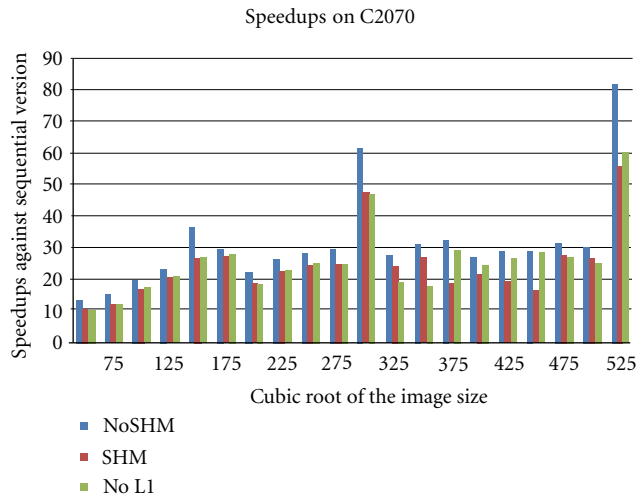


Figure 9: The overall speed up of the algorithm on the C2070 system for different image sizes (with the data transfer times excluded).

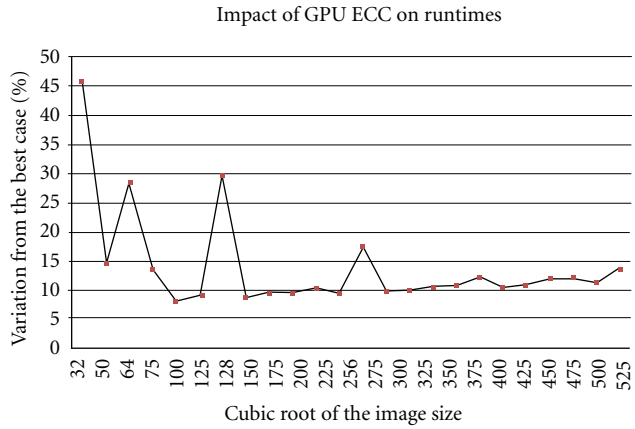


Figure 10: The variation of the runtime when ECC is enabled as a percentage of the best case for the C2070 system.

associated with a large number of check bits. To evaluate the impact of ECC, we ran the algorithm with and without ECC. To avoid any direct influence of L1 on this, we turned off the level-1 cache. We report the variation of the runtimes as a percentage of the best case, the non-ECC version, in Figure 10. However, in our case, we did not see any difference in accumulated errors, which is a random event. As can be observed, the variation diminishes as image sizes increases but is more pronounced at the power-of-two problem sizes. At the power-of-two problem sizes, the number of check bits required for ECC is higher than for nonpower-of-two problem sizes [17], and thus overheads are considerably high. We assume that with increasing data size, the accesses are more consolidated to blocks, and thus the number of separate checks reduces.

6. Conclusions

In this paper, we investigated the mapping of an enhanced motion estimation algorithm to a number of GPU-specific

architectures, resulting challenges and benefits therein. Using a database of three-dimensional image sequences, we showed that the mapping leads to substantial performance gains, up to a factor of 60, and can provide near-real-time experience. By doing this, we gained more insight into the process. From our investigation, we observed the following.

(i) Although the presence of different memory subsystems are key in GPU programming, their significance is diminishing. We witnessed this simply with the use of shared and constant memories against level-1 and level-2 caches. Partly, this observation is very influential across image processing applications—where working with large amounts of data is a fundamental requirement. In modern Fermi-based systems, the loading resolution of the cache controllers amortizes the overheads in managing different memory subsystems.

(ii) In three-dimensional image processing applications, the spatial locality can only be exploited along one dimension in the CPU, while there is no spatial locality in the other two dimensions. This leads to benefits along one of the dimensions on the CPU. Meanwhile, on the GPU-front, nonspatially local dimensions benefit from coalesced memory access. However, memory accesses along the remaining dimension do not benefit from coalesced memory access. These two facts are difficult to assess without detailed profiling but are evident in the fact that coalesced access leads to better performance.

(iii) In three-dimensional image processing applications, increasing the number of smoothing iterations on the GPU will not change the overall speedup although it increases the absolute runtimes.

(iv) Even in the absence of the level-1 (L1) cache, the performance was sustained by the level-2 cache.

(v) Error correction and checking (ECC) is necessary for reliable outputs. However, wherever possible, this can be traded off for performance.

(vi) In a typical image processing application, the motion estimation pipeline is repeatedly applied for subsequent frames of image sequences, and therefore, it is valid to assume that the data will persist on the device and long-term runtime benefits will amortize the cost of host-to-device and device-to-host transfers.

(vii) The overall cost per performance is very attractive. The cost of a 512-core GPU is only a quarter of a 16-core CPU-based system (as of mid 2011). Nevertheless, the GPU-based system yields noticeable performance benefits. Furthermore, the evolving and simplified architectural and programming models, makes this an excellent option for biomedical image processing applications.

Although our work has exploited several different aspects of the GPU-architecture, there are several different aspects which may be improved.

(i) For extremely large datasets, where the device memory cannot hold the entire dataset, it may be necessary to perform distributed processing using multiple GPUs. Although both our algorithm and the software framework can easily be extended to cover this, the immediate benefits on near-real-time experience is not known.

(ii) The current Fermi-based GPUs support concurrent kernel execution, which permits launching multiple kernels in a concurrent fashion. This feature essentially liberates the GPU and unlocks it from traditional SIMD-style processing. The exact benefits still need to be investigate especially in the context of three-dimensional image processing applications.

(iii) To alleviate the intricacies relating to conditionals, the current version of the algorithm does a fixed number of iterations. A more suitable method is needed to adaptively control the convergence rate.

With all these, we find that although exploiting architectural peculiarities rendered tangible benefits, these benefits are narrowing with the evolving architecture and simplified programming models. The future of GPU architectures will incorporate prefetching [18] and will support abstractions at the higher level.

Acknowledgments

The authors would like to thank Professor Mike Giles from the Mathematics Institute, University of Oxford, and Jing Guo from the University of Hertfordshire for their invaluable inputs. They also would like to thank the anonymous reviewers for their feedback and comments on the work presented in this paper. This research is supported by the Oxford Martin School, University of Oxford. V. Grau is supported by an RCUK Academic Fellowship.

References

[1] Khronos Group: The OpenCL Specification 1.1 (Last accessed May 20, 2011) http://www.khronos.org/opencl/.

[2] NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture Programming Guide 3.0 (June 2010).

[3] D. B. Kirk and H. W. Wen-mei, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kauffmann, Boston, Mass, USA, 1st edition, 2010.

[4] J. Marzat, Y. Dumortier, and A. Ducrot, "Real-time dense and accurate parallel optical flow using CUDA," in *Proceedings of the 17th International Conference in Central Europeon Computer Graphics, Visualization and Computer Vision (WSCG '09)*, 2009.

[5] D. L. G. Hill, P. G. Batchelor, M. Holden, and D. J. Hawkes, "Medical image registration," *Physics in Medicine and Biology*, vol. 46, no. 3, pp. R1–R45, 2001.

[6] J. B. A. Maintz and M. A. Viergever, "A survey of medical image registration," *Medical Image Analysis*, vol. 2, no. 1, pp. 1–36, 1998.

[7] B. Zitová and J. Flusser, "Image registration methods: a survey," *Image and Vision Computing*, vol. 21, no. 11, pp. 977–1000, 2003.

[8] T. Mäkelä, P. Clarysse, O. Sipilä et al., "A review of cardiac image registration methods," *IEEE Transactions on Medical Imaging*, vol. 21, no. 9, pp. 1011–1021, 2002.

[9] A. Bruhn, J. Weickert, and C. Schnrr, "Lucas/Kanade meets Horn/Schunck: combining local and global optic flow methods," *International Journal of Computer Vision*, vol. 61, pp. 211–231, 2005.

[10] B. K. P. Horn and B. G. Schunck, *Determining Optical Flow. Technical report*, Massachusetts Institute of Technology, Cambridge, Mass, USA, 1980.

[11] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pp. 674–679, 1981.

[12] A. Bruhn, J. Weickert, T. Kohlberger, and C. Schnörr, "A multigrid platform for real-time motion computation with discontinuity-preserving variational methods," *International Journal of Computer Vision*, vol. 70, no. 3, pp. 257–277, 2006.

[13] F. Yin, C. Chan, and R. Judd, "Compressibility of perfused passive myocardium," *American Journal of Physilogy*, vol. 271, no. 5, pp. H1864–H1870, 1996.

[14] F. Yin, C. Chan, and R. Judd, "Compressibility of perfused passive myocardium," *IEEE Transactions on Medical Imaging*, vol. 271, no. 8, pp. H1864–H1870, 1996.

[15] S. M. Song and R. M. Leahy, "Computation of 3-D velocity fields from 3-D cine CT images of a human heart," *IEEE Transactions on Medical Imaging*, vol. 10, no. 3, pp. 295–306, 1991.

[16] R. Damelio, *Basics of Benchmarking*, Productivity Press, 1st edition, 1995.

[17] R. W. Hamming, "Error detecting and error correcting codes," *Bell System Technical Journal*, vol. 26, no. 2, pp. 147–160, 1950.

[18] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kauffmann, Boston, Mass, USA, 3rd edition, 2002.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 572187, 16 pages
doi:10.1155/2011/572187

Research Article

Efficient Probabilistic and Geometric Anatomical Mapping Using Particle Mesh Approximation on GPUs

Linh Ha,[1] Marcel Prastawa,[1] Guido Gerig,[1] John H. Gilmore,[2] Cláudio T. Silva,[1] and Sarang Joshi[1]

[1] *Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT 84112, USA*
[2] *Department of Psychiatry, University of North Carolina, Chapel Hill, NC 27599, USA*

Correspondence should be addressed to Linh Ha, lha@sci.utah.edu

Received 2 March 2011; Revised 6 May 2011; Accepted 3 June 2011

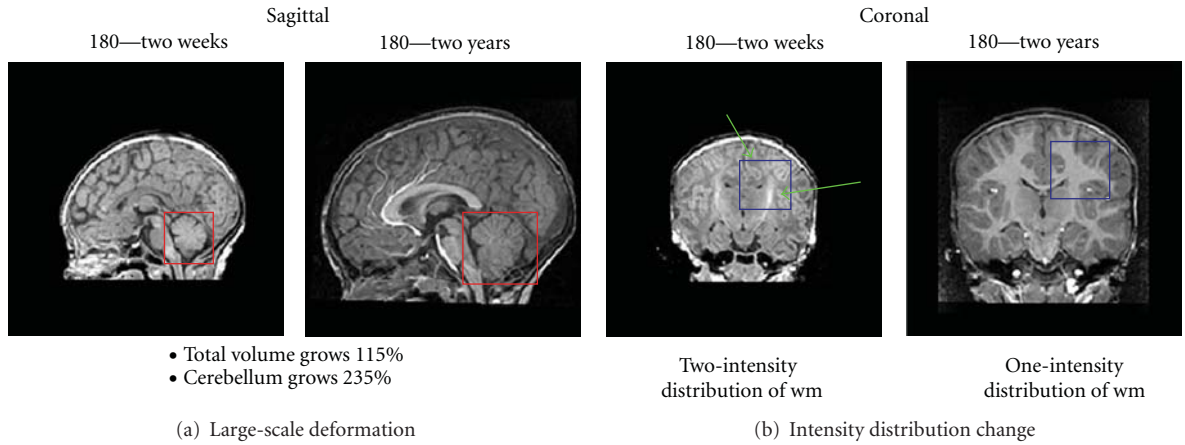
Academic Editor: Aly A. Farag

Deformable image registration in the presence of considerable contrast differences and large size and shape changes presents significant research challenges. First, it requires a robust registration framework that does not depend on intensity measurements and can handle large nonlinear shape variations. Second, it involves the expensive computation of nonlinear deformations with high degrees of freedom. Often it takes a significant amount of computation time and thus becomes infeasible for practical purposes. In this paper, we present a solution based on two key ideas: a new registration method that generates a mapping between anatomies represented as a multicompartment model of class posterior images and geometries and an implementation of the algorithm using particle mesh approximation on Graphical Processing Units (GPUs) to fulfill the computational requirements. We show results on the registrations of neonatal to 2-year old infant MRIs. Quantitative validation demonstrates that our proposed method generates registrations that better maintain the consistency of anatomical structures over time and provides transformations that better preserve structures undergoing large deformations than transformations obtained by standard intensity-only registration. We also achieve the speedup of three orders of magnitudes compared to a CPU reference implementation, making it possible to use the technique in time-critical applications.

1. Introduction

Our work is motivated by the longitudinal study of early brain development in neuroimaging, which is essential to predict the neurological disorders in early stages. The study, however, is challenging due to two primary reasons: the large-scale nonlinear shape changes (the image processing challenge) and the huge amount of computational power the problem requires (the computational challenge). The image processing challenge involves robust image registration to define anatomical mappings. While robust image registrations have been studied extensively in the literature [1–3], registration of the brain at early development stage is still challenging as the growth process can involve very large-scale size and shape changes, as well as changes in tissue properties and appearance (Figure 1). Knickmeyer et al. [4] showed that the brain volume grows by 100% the first year and 15% the second year, whereas the cerebellum shows 220% volume growth for the first and another 15% for the second year. These numbers indicate very different growth rates of different anatomical structures. Through regression on shape representations, Datar et al. [5] illustrated that the rapid volume changes are also paralleled by significant shape changes, which describe the dynamic pattern of localized, nonlinear growth. A major clinical research question is to find a link between cognitive development and the rapid, locally varying growth of specific anatomical structures. This requires registration methods to handle large-scale and also nonlinear changes. Also, the process of white matter myelination, which manifests as two distinct white matter appearance patterns primarily during the first year of development, imposes another significant challenge as image intensities need to be interpreted differently at different stages.

To approach these problems, a robust registration method is necessary for mapping longitudinal brain MRI to



(a) Large-scale deformation (b) Intensity distribution change

FIGURE 1: Registration challenges of human brains at early development stages. The image show significant shape and size changes of an infant brain of subject 180 from two weeks to two years as well as the changing white matter properties and appearance due to the myelination.

a reference space so that we can perform reliable analysis of the tissue property changes reflected in MR measurements. This method should not rely on raw intensity measurements, while it should be capable of estimating large structural deformations. Xue et al. [6] addressed these issues by proposing a registration scheme for neonatal brains by registering inflated cortical surfaces extracted from the MRI. Their registration method does not make use of voxel-wise image information and is not intended to capture growth in internal structures. It is designed for analyzing cortical surfaces, and it does not define a transformation for the whole brain volume.

In this paper, we propose a new registration framework for longitudinal brain MRI that makes use of underlying anatomies, which are represented by geometries and class posterior images. This framework can match internal regions and simultaneously preserve a consistent mapping for the boundaries of relevant anatomical objects. We show results of registering neonatal brain MRI to 2-year old brain MRI of the same subjects obtained in a longitudinal neuroimaging study. Our method consistently provides transformations that better preserve time-varying structures than those obtained by intensity-only registration [7].

The study presents a significant computational challenge because dense, free-form mapping is computationally expensive. In particular, a correspondence-free geometric norm such as "currents" has computational complexity of $O(M^2)$ where M is the number of geometric elements, which is in the same order of the image volume [8]. These methods require supercomputing power to run [9], but still take a considerable amount of time to complete. While access to a supercomputer system or even a cluster is not available to most researchers, robust registration in the presence of large deformations is essential. Fortunately, this computation problem finds an economical solution via the work of High-Performance Computing (HPC) General Processing on Graphical Processing Units (GPUs) community. Modern GPUs, which are available on commodity hardware, could offer several teraflops of peak performance, which is equivalent to that of a super computer in the mid-90s. There have been a number of image processing applications being implemented on GPUs [10–13]. Most applications achieve from 20x to several magnitudes of speedup when moved to GPUs in comparison to conventional CPU versions. A closely related example is the fast Greedy Iterative Diffeomorphic registration framework by Ha et al. [14] using GPUs that achieved 60x speedup in comparison to an optimized, fully parallel version running on an eight-core Xeon 3.2 Ghz sever.

However, mapping algorithms from the CPU to the GPU is nontrivial. The GPU programming model is significantly different from the CPU programming model. While GPUs are highly efficient for parallel data processing, they are slow for serial scalar code, which exists in any processing algorithms. To achieve a high performance, it often requires developers to reformulate the problem so that it is mapped well to the GPU architecture. In this paper, we present the implementation of our registration framework on commodity GPUs. We introduce two primary performance improvements with a combination of two approaches: (1) an algorithmic improvement using a particle mesh approach and (2) parallelisation using GPUs. We are able to solve the practical problem in real time and gain speedup of nearly three magnitudes order over CPU reference implementation.

2. Related Work

The development of image registration is the major focus of computational anatomy [3, 15–17]. There are two large bodies of research that our method is developed on: large deformation diffeomorphic registration and multicompartment registration via surface matching.

The analysis of shape and size in anatomical images models anatomy as a deformable template [18]. Common image registration techniques based on thin-plate splines and linear-elastic models [19, 20] have a small deformation assumption and cannot be used due to the large localized

deformations associated with early brain development. The large deformation model for computing transformations developed by Christensen et al. [21] overcomes the limitations of the small deformations model by ensuring that the transformations computed between imagery are diffeomorphic (smooth and invertible). Based on the large deformation framework by Miller and Younes [3], Beg et al. [22] derived the Large Deformation Diffeomorphic Metric Mapping (LDDMM) algorithm. This method computes an optimal velocity field that satisfies the Euler-Lagrange variational minimization constraints. Our method is developed upon the greedy approach proposed by Christensen et al. [21] that often reports high registration quality comparable to LDDMM approach but requires significantly lower amount of computation.

Surface matching is usually considered a semiautomatic procedure and a "point correspondence" task. First, a small number of anatomical features such as landmark points and curve are identified by hand. Next, each of these features of the discretized surface finds its corresponding feature on the target. This matching information is then used to guide the transformation of the entire surface [19, 23, 24]. This approach, however, has a fundamental issue due to discretization. The currents distance was introduced by Vaillant and Glaunès [25] as a way of comparing shapes (point sets, curves, surfaces) without having to rely on computing correspondences between features in each shape.

Most of the current registration techniques currently being used in computational anatomy are based on single-subject anatomy [18, 25–27]. This approach is limited since a single anatomy cannot faithfully represent the complex structural variability and development of the subjects. Our method is based on the multicompartment model proposed by Glaunes and Joshi [28] which defines a combined measurement acting on different anatomical features such as point, curve, and surface to enhance registration quality.

Existing works refer to computational anatomy, especially free-from matching, as a robust but computationally expensive framework which is difficult to achieve in real time on commodity hardware [9, 29, 30]. In this paper, we consider GPU implementation as an integral part of our work and an essential contribution that allows scientists to accurately register images and geometries in time-critical applications.

3. Method

We propose a new registration method that makes use of the underlying anatomy in the MR images. Figure 2 shows an overview of the registration process. We begin by extracting probabilistic and geometric anatomical descriptors from the images, followed by computing a transformation that minimizes the distance between the anatomical descriptors.

3.1. Anatomical Descriptors. We represent brain anatomy as a multicompartment model of tissue class posteriors and manifolds. We associate each position x with a vector of tissue probability densities. In a given anatomy, we capture the underlying structures by estimating, for each image, the class posterior mass functions associated with each of the classes. Given Ω as the underlying coordinate system of the brain anatomies, each anatomy $\mathcal{A}_{i=1,\ldots,N}$ is represented as

$$\mathcal{A}_i = \left\{ p_{i,c=1}(x), \ldots, p_{i,c=N_c}(x), \mathcal{M}_{i,j=1}(2), \ldots, \mathcal{M}_{i,j=N_s}(2) \subset \Omega \right\}, \tag{1}$$

where N_c is the number of probability images, N_s is the number of surfaces, $p_c(x)$ is the class posterior for tissue c at location x, and $\mathcal{M}_j(2)$ are 2-dimensional submanifolds of Ω (surfaces).

As we are interested in capturing major growth of the white matter and gray matter growth, we represent brain anatomy as a tuple of the probabilities $\{p_{\mathrm{wm}}(x), p_{\mathrm{gm}\}(x),\mathrm{p}_{\mathrm{csf}}}(x)\}$ representing class posterior probabilities of white matter, gray matter, and cerebrospinal fluid respectively, followed by the surfaces of white matter, gray matter, and cerebellum.

The classification of brain MR images with mature white matter structures into class posteriors is well studied. We extract the posteriors from 2-year old brain MR images using the segmentation method proposed by van Leemput et al. [31]. The method generates posterior probabilities for white matter (wm), gray matter (gm), and cerebrospinal fluid (csf). These probabilities can then be used to generate surfaces from the maximum a posteriori tissue label maps.

The classification of neonatal brain MR images is challenging as the white matter structure undergoes myelination, where the fibers are being covered in myelin sheathes. Several researchers have proposed methods that make use of prior information from an atlas or template that takes into account the special white matter appearance due to myelination [32]. We use the method described by Prastawa et al. [33] for extracting the tissue class posteriors of neonatal brain MRI, which includes for myelinated wm, nonmyelinated wm, gm, and csf. These can then be used to create an equivalent anatomy to the 2-year old brain by combining the two white matter class probabilities which then leads to a single white matter surface.

The white matter and gray matter surfaces are generated from the maximum a posteriori (MAP) segmentation label maps using the marching cubes algorithm [34]. The cerebellum surfaces are generated from semiautomated segmentations that are obtained by affinely registering a template image followed by a supervised level set segmentation. The cerebellum has a significant role in motor function, and it is explicitly modeled as it undergoes the most rapid volume change during the first year of development and thus presents a localized large-scale deformation.

3.2. Registration Formulation. Given two anatomies $\mathcal{A}_1$ and $\mathcal{A}_2$, the registration problem can be formulated as an estimation problem for the transformation h that minimizes

$$\hat{h} = \underset{h}{\operatorname{argmin}}\, E(h \cdot \mathcal{A}_1, \mathcal{A}_2)^2 + D(h, e)^2 \,, \tag{2}$$

where $h \cdot \mathcal{A}_1$ is the transformed anatomy, $E(\cdot, \cdot)$ is a metric between anatomies, and $D(\cdot, e)$ is a metric on a group of transformations that penalizes deviations from

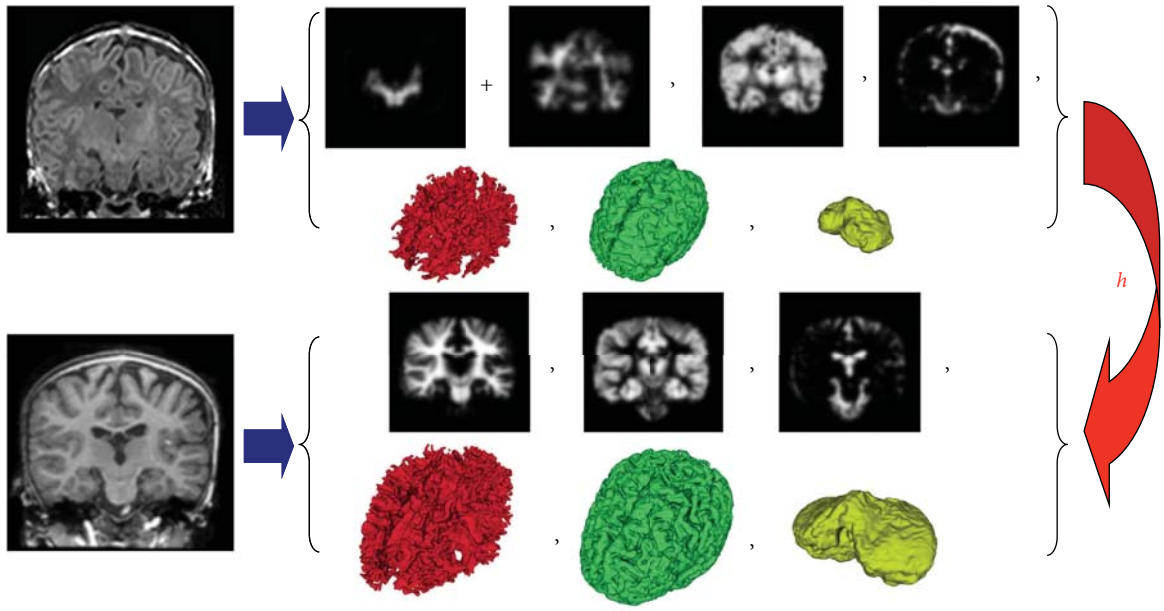


Figure 2: Overview of the proposed registration method that can handle large deformations and different contrast properties, applied to mapping brain MRI of neonates to 2-year olds. We segment the brain MRIs and then extract equivalent anatomical descriptors by merging the two different white matter types present in neonates. The probabilistic and geometric anatomical descriptors are then used to compute the transformation h that minimizes the distance between the class posterior images, as well as the distance between surfaces represented as currents.

the identity transformation e. The anatomy is transformed using backward mapping for probability image and forward mapping for geometries:

$$\begin{aligned} h \cdot \mathcal{A}_1 = h \cdot \Big\{ & p_{i,c=1}(x), \ldots, p_{i,c=N_c}(x), \\ & \mathcal{M}_{i,j=1}(2), \ldots, \mathcal{M}_{i,j=N_s}(2) \Big\} \\ = \Big\{ & p_{i,c=1}(x) \circ h^{-1}, \ldots, p_{i,c=N_c}(x) \circ h^{-1}, \\ & h\Big(\mathcal{M}_{i,j=1}(2)\Big), \ldots, h\Big(\mathcal{M}_{i,j=N_s}(2)\Big) \Big\}. \end{aligned} \tag{3}$$

We define distance between anatomies E by defining a norm on an anatomy as a combination of the L^2 norm on the class posteriors and a Reproducing Kernel Hilbert space norm on the manifolds defined as "currents" through Glaunes et al. [1]. This norm does not require prior knowledge on geometric correspondence, as compared to other geometry matching methods [19, 23, 24] that require explicit specification of geometric correspondences. More precisely, they require that a certain point q in object A is the same (anatomically) as point q in object B; hence, object A and object B are required to have the same number of elements and the same ordering of elements. In comparison, the currents norm defines distance between objects based on the norm measurement of the union of the geometric objects. The currents norm is thus correspondence-free and does not require the objects in comparison to have equal number of elements and the same ordering or anatomical definition.

In contrast to Iterative Closest Point (ICP) algorithm [35] which defines correspondence based on the closest features on the Euclidean space, the currents matching algorithm compares each element to all other elements. Since there may not exist an anatomically homologous correspondence for every feature due to discretization, the currents matching is more robust than existing methods. Manifolds with different number of elements (resolutions) can thus be matched using the currents norm due to this property. For an oriented surface $\mathcal{M}(2)$ in R^3 the norm $[\mathcal{M}(2)]$ is the vector-valued Borel measure corresponding to the collection of unit normal vectors to $\mathcal{M}(2)$, distributed with density equal to the element of surface area ds and can be written as $\eta(x)ds(x)$, where $\eta(x)$ is the unit normal and $ds(x)$ is the surface measure at point x. The currents representation forms a vector space that admits linear operations, unlike other surface representations such as the Signed Distance Map [36–38].

Given an anatomy $\mathcal{A}$ the k-norm of $[\mathcal{A}]$ is composed as

$$\|[\mathcal{A}]\|_k^2 = \|P(x)\|_{L^2} + \|[\mathcal{M}(2)]\|_k, \tag{4}$$

where the probabilistic norm is defined as

$$\begin{aligned} \|P(x)\|_{L^2} &= \sum_{c=1}^{N_c} \left\| p_{1,c}(x) - p_{2,c}(x) \right\|_k^{L^2} \\ &= \int_\Omega |p_{1,c}(x) - p_{2,c}(x)|^2 dx \end{aligned} \tag{5}$$

and the currents norm is given by

$$\|[\mathcal{M}(2)]\|_k = \iint_{\mathcal{M}(2)} k(x,y)\langle \eta(x), \eta(y)\rangle d\mu(x)d\mu(y), \tag{6}$$

where $k(\cdot,\cdot)$ is a shift-invariant kernel (e.g., Gaussian or Cauchy).

When $\mathcal{M}(2)$ is a discrete triangular mesh with N_f faces, a good approximation of the norm can be computed by replacing $[\mathcal{M}(2)]$ by a sum of vector-valued Dirac masses

$$\|[\mathcal{M}(2)]\|_k^2 = \sum_{f=1}^{N_f}\sum_{f'=1}^{N_f} \langle \eta(f), \eta(f')\rangle k(c(f), c(f')), \tag{7}$$

where N_f is the number of faces of the triangulation and, for any face f, $c(f)$ is its center and $\eta(f)$ its normal vector with the length capturing the area of each triangle.

Having defined the norm on probability images and surfaces, the dissimilarity metric between anatomies $\|[\mathcal{A}_1] - [\mathcal{A}_2]\|_k^2$ is given by

$$\begin{aligned} &w_p \sum_{c=1}^{N_c} \|p_{1,c}(x) - p_{2,c}(x)\|_k^{L^2} + w_g \sum_{j=1}^{N_s} \left\| \left[\mathcal{M}_{1,j}(2) - \mathcal{M}_{2,j}(2)\right] \right\|_k^2 \\ &\quad = w_p \sum_{c=1}^{N_c} \int_\Omega |\, p_{1,c}(x) - p_{2,c}(x)\,|^2 dx \\ &\qquad + w_g \sum_{j=1}^{N_s} \left\| \left[\mathcal{M}_{1,j}(2) \cup \left(-\mathcal{M}_{2,j}(2)\right)\right] \right\|_k^2, \end{aligned} \tag{8}$$

where the distance between two surface currents $\|[\mathcal{M}_{1,j}(2) - \mathcal{M}_{2,j}(2)]\|_k = \|[\mathcal{M}_1(2) \cup (-\mathcal{M}_2(2))]\|_k$ is computed as the norm of the union between surface $\mathcal{M}_1(2)$ and surface $\mathcal{M}_2(2)$ with negative measures, w_p and w_g are scalar weights that balance the influence of probabilistic and geometric presentations.

We use the large deformation framework [3] that generates dense deformation maps in R^d by integrating time-dependent velocity fields. The flow equation is given by $\partial h^v(t,x)/\partial t = v(t, h^v(t,x))$, with $h(0,x) = x$, and we define $h(x) := h^v(1,x)$, which is a one-to-one map in $\mathbb{R}^d$, that is, a diffeomorphism. The diffeomorphism is constructed as a fluid flow that is smooth and invertible. The invertibility of the mapping is a desirable property as it enables analysis in different spaces and time points as needed. We define an energy functional that ensures the regularity of the transformations on the velocity fields: $\|v(t,\cdot)\|_V^2 = \int_{\mathbb{R}^d} \langle Lv(t,x), Lv(t,x)\rangle dx$, where L is a differential operator acting on vector fields. This energy also defines a distance in the group of diffeomorphisms:

$$D^2(h,e) = \inf_{v, p^v(1,\cdot)=h} \int_0^1 \|Lv(t)\|_V^2 dt. \tag{9}$$

The registration optimizations in this paper are performed using a greedy approach by iteratively performing gradient descent on velocity fields and updating the transformations via an Euler integration of the O.D.E. At each iteration of the algorithm the velocity field is calculated by solving the PDE:

$$Lv = F(h), \tag{10}$$

where v is the transformation velocity field, $L = \alpha\nabla^2 + \beta\nabla\cdot\nabla + \gamma$, and $F(h)$ is the variation of $\|[h\cdot\mathcal{A}_1] - [\mathcal{A}_2]\|_k^2$ with respect to h. This variation is a combination of the variation of the L^2 norm on the class posteriors and of the currents norm, computed using the gradient

$$\begin{aligned} \frac{\partial \|[\mathcal{M}(2)]\|_k^2}{\partial x_r} &= \sum_{f|x_r\in f} \left[\frac{\partial \eta(f)}{\partial x_r}\right] \sum_{f'=1}^{N_f} k(c(f'), c(f))\eta(f') \\ &\quad + \frac{2}{3}\sum_{f'=1}^{N_f} \frac{\partial k(c(f), c(f'))}{\partial c(f)} \eta(f')^t \eta(f), \end{aligned} \tag{11}$$

given that points $\{x_r, x_s, x_t\}$ form the triangular face f and its center $c(f) = (x_r + x_s + x_t)/3$ and its area-weighted normal $\eta(f) = (1/2)(x_s - x_r)\otimes(x_t - x_r)$.

The currents representation is generalized to account for not only surface meshes but also other m-submanifolds such as point sets or curves. The currents associated to an oriented m-submanifold $\mathcal{M}$ is the linear functional $[\mathcal{M}]$ defined by $[\mathcal{M}](\omega) = \int_{\mathcal{M}} \omega$. When $\mathcal{M}(0) = \bigcup x_i$ is a collection of points $[\mathcal{M}(0)]$ is a set of Dirac delta measures centered at the points that is, $[\mathcal{M}(0)] = \sum_i \alpha_i \delta(x - x_i)$. When $\mathcal{M}(1)$ is a curve in $\mathbb{R}^3$, $[\mathcal{M}(1)]$ is the vector-valued Borel measure corresponding to the collection of unit-tangent vectors to the curve, distributed with density equal to the element of length dl:

$$\|[\mathcal{M}(1)]\|_k^2 = \sum_{l=1}^{N_l}\sum_{l'=1}^{N_l} \langle \tau(l), \tau(l')\rangle k(c(l), c(l')), \tag{12}$$

where N_l is the number of line segments and, for any segment l with vertices v_0 and v_1, $c(l) = (v_o + v_1)/2$ is its center and $\tau(l) = v_1 - v_0$ is its tangent vertor with its length capturing the length of the line segment.

Using extra submanifold presentation helps capture important properties of the target anatomy and hence could potentially direct the registration and improve the result; see Glaunes et al. [1] for more details.

4. Efficient Implementation

The implementation of our registration framework is based on two critical sections: large deformation diffeomorphic image registration and currents norm computation. The former requires a linear solver (10) on an $M \times M$ matrix where M is the number of input volume elements ($\approx$10 millions on typical brain image). The linear system is sparse and there exists efficient solver with complexity of $O(M\log(M))$. The performance is even further amortized using a multiscale iterative method resembling a multigrid

solver. The method maps well to the GPU architecture and significantly reduces the running time from several hours on eight-core sever to a few minutes on commodity hardware. We refer to the work by Ha et al. [14] for details of the method and implementation of large deformation diffeomorphic registration on GPUs. Here, we concentrate on the problem of how to implement norm computation efficiently based on GPU methodologies.

At a broad level, the GPUs consist of several streaming multiprocessors—each of them contains a number of streaming processors and a small shared memory unit. GPUs are good at handling data stream in parallel with processing kernels [39]. The underlying program structure is described by streams of data passing through computation kernels. Given a set of data (an input stream), a series of operations (kernel functions) are applied to each element in the stream and produce another set of output data (an output stream). The program is constructed by chaining these computations together. This formulation has been used to design efficient GPU-based sorting and numerical computations [14, 40, 41].

4.1. Particle Mesh Approximation for Currents Norm Computation. The major challenge of computing the currents norm (7) for real brain surfaces is the high computational cost to compute the dissimilarity metric of all pairs of surface elements, which is $O(N_f^2)$, where N_f is the number of faces. A surface extracted from an N^3 volume has the average complexity of $N^{2.46}$ faces [8], that produces millions surfaces for a typical 256^3 input.

For computational tractability, Durrleman et al. [42] used a sparse representation of the surface based on matching pursuit algorithm. On the other hand, an efficient framework based on the standard fast Gauss transform [43] requires the construction and maintenance of the kd-tree structures on the fly. The primary problem of these approaches is that while the performance is insufficient for real-time applications on conventional systems, they are too sophisticated to make use of processing power of modern parallel computing models on GPUs. Also in practice, we use large kernel width for the currents norm to match major structures. This is not ideal for kd-tree-based implementations that are designed for querying small set of nearest neighbor. Implementing these ideas on GPUs imposes other challenges, and they are unlikely to be efficient.

Here, we employ a more parallelizable approach based on the Particle Mesh approximation (PM). This approximation has been extensively studied in a closely related problem—the cosmological N-body simulation, which requires the computation of the interaction between every single pair of objects (see Hockney and Eastwood [44] for details).

The particle mesh approximation, as shown in Figure 3, includes four main steps.

Grid building which determines the discretization error or the accuracy of the approximation. It also specifies the computational grid, the spacial constraints of the computation. The quantization step in each spacial direction determines the grid size, hence, the complexity of the grid computation. The finer the grid means the higher quality of the approximation but the more computation involving.

Splatting that maps computation from an unstructured grid to a structured grid. It is the inverse operation of the interpolation.

Integration which performs the grid computation and updating step. As the computation, which involves kernel convolution and gradient computation, is taking place in a regular domain, the integration can exploit the parallel processing power of special computing units such as GPUs.

Interpolation that interprets computational results from the image space back to the geometrical space, in other words, to reconstruct the unstructured grid out of the structured domain. Marching Cube [34] is an example of techniques using interpolation to extract isosurfaces from MR images.

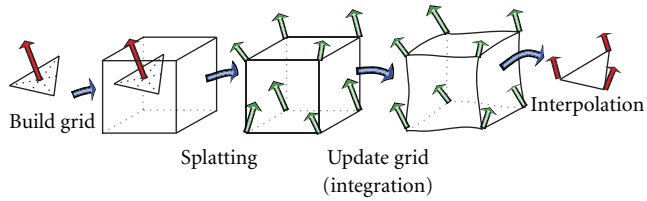


Figure 3: Particle mesh approximation algorithm to transform the computation from irregular domain to regular domain based on four basic steps: grid construction, splatting, integration, and interpolation.

The splatting/interpolation operation pair works as a connection between the computation on regular domain and irregular domain. We will go into details of how to implement this interface on the parallel architecture as the method can be widely used not only for the norm computation but any mixed—geometric and probabilistic—computation in general. We consider this strategy as a crucial method for efficient parallel computation on an irregular domain.

The error in particle mesh approximation is influenced by two factors: the grid spacing and the width of the convolution kernel, as shown in Figure 4. We chose the image grid spacing, thus the error is bounded by the image resolution. As being aforementioned, we use large kernel widths in practice which is ideal for PM. Note that PM approximation breaks down when kernel width is less than grid spacing.

While the approximation helps reduce the complexity to $M \log M$ where M is the volume size of the embedded grid, the total complexity of the method is still very high. On a high-end workstation with 8-CPU cores, a highly optimized multithreaded implementation in C++ takes several hours for one matching pair hence cannot be used for parameter exploration and real-time analysis. Based on the GPU framework by Ha et al. [14], we developed an

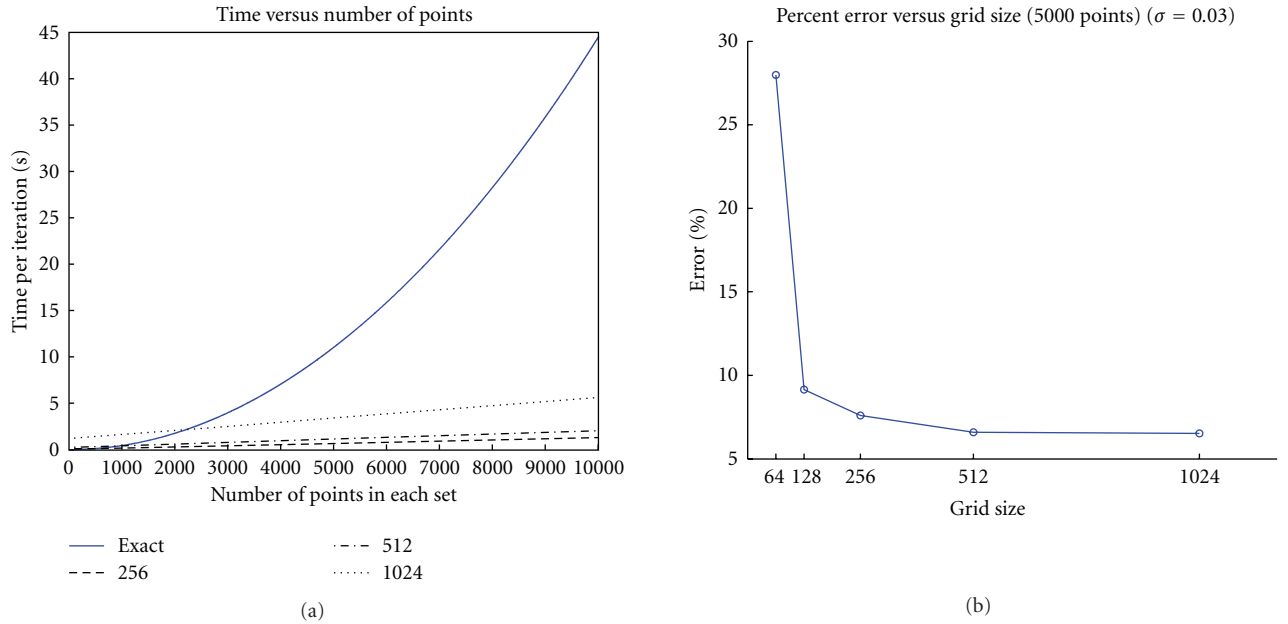


Figure 4: (a) shows the run time comparisons between direct computation and the particle mesh implementation for various grid size. Shown in (b) is the percent error for different for 5000 randomly generated points with different mesh sizes.

implementation that runs entirely on the GPU to exploit parallel efficiency of regular grid presentation.

4.2. Efficient Implementation of Particle Mesh Method on GPUs. To achieve the maximum performance efficiency, we optimized the four steps of particle mesh method on GPUs. Here, we describe the performance keys and important details to implement these steps.

4.2.1. Grid Building. Without prior information, computational grid is typically chosen as a discretization of the bounding box with extra border regions to prevent out-of-bound quantization error. Since probabilistic and geometric descriptors coexist in our representation, the computational grid is effectively chosen as the original grid. This selection guarantees that it will not introduce further quantization errors than the original discretized errors inherent to the construction of geometric descriptors. This strategy also limits the complexity of the combining technique to the original order of computation if we use only probabilistic terms.

4.2.2. Splatting. The main purpose of the splatting function is to construct a regular n-dimensional scalar or vector field from its discrete sample points. The constructed grid should satisfy an inverse operation, the interpolation, so that when applied to the reconstructed grid will reproduce the sample points. In other words, Interpolation (Splatting $(E)) = E$ with E is an arbitrary input. This duality of splatting and interpolation reflects the fact that probabilistic and geometry descriptors are just the domain representations of the same subject. Hence, we could unify their computation without losing accuracy. We also exploit the duality to validate the correctness of our implementation of the splatting function through its dual counterpart.

The splatting function is defined by Trouvé and Younes [45] through a linear operator $\aleph$ that applies a mapping vector field $v : \mathbb{Z}^d \to \mathbb{R}$ to a discrete image $I : \mathbb{Z}^d \to R$ to perform an interpolation on the grid $G_v = \{x+v(x) | x \in \mathbb{Z}^d\}$, mathematically saying

$$(\aleph I)(x) = (\Im)(x + v(x)), \tag{13}$$

with $\Im$ being linear interpolation, defined by

$$(\Im)(I)(x) = \sum_{\epsilon \in \{0,1\}^d} c_\epsilon(x) I(\lfloor x_1 \rfloor + \epsilon_1, \lfloor x_2 \rfloor + \epsilon_2, \ldots, \lfloor x_d \rfloor + \epsilon_d), \tag{14}$$

with $\lfloor z \rfloor$ being the integer part of real number z and $\{z\} = z - \lfloor z \rfloor$ is the fractional part. The coefficient $c_\epsilon(x)$ is defined as

$$c_\epsilon(x) = \prod_{i=1}^{d} (\epsilon_i + (1 - 2\epsilon_i) x_i). \tag{15}$$

While the splatting operator was defined through a vector field, the splatting conversion from the irregular grid to the regular domain for an arbitrary input is defined as being a zero vector field. Figure 5 displays the construction of a regular grid presentation of geometrical descriptors in 2D through splatting operator. The value at a grid point is computed by accumulating values interpolated at that point from its geometrical neighbors. Thus, closer neighbors will have more influence on the value of the point than farther points. In fact, we only need to consider the one-ring neighbors as farther points have a negligible contribution to

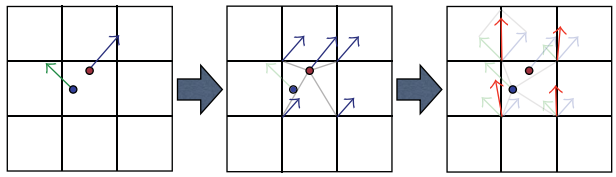


Figure 5: Geometrical conversion based on a splatting function with zero velocity field v (13). The method served as a bridge to transform the computation from an irregular grid to a regular grid which allows an efficient parallel implementation.

its final value. We also assume that the field is continuous and smooth.

Though the splatting operator has a linear complexity in terms of the size of geometry descriptors, it is the performance bottleneck in practice. The single CPU thread-based splatting function is too slow for interactive applications. Even close discrete points do not share the same cache as the definition of a neighbor in 3D does not map to a neighbor in the linear CPU cache. The multithread-based CPU splatting, which assigns each thread a single geometrical element, however, has a resource-fighting problem. That is, when we integrate grid value from its neighbor submanifold elements, it is likely that there are several elements in the neighbor, and these elements, which are assigned different threads, may try to accumulate the grid value at the same time. GPU implementation also has to face with the resource-fighting problem.

We can apply mutex locking to resolve the conflict. However, it is inefficient with thousands of threads on GPUs. A better solution is based on atomic operations, which are guaranteed to complete without being interrupted by the actions of other threads. Currently, CUDA does not support atomic operations for floating point numbers but integer numbers. Here we propose two different approaches for splatting computation: the collision-free splatting scheme via a fast parallel sorting and the atomic splatting scheme using a fixed-point representation.

The *collision-free splatting* scheme is applied for systems without any atomic operation support. As shown in Figure 6, we employ a fast parallel sorting to resolve the shared-resource fighting problem. The algorithm involves three steps.

(i) Compute the contribution of each geometrical descriptor to grid nodes.

(ii) Sort the contribution based on node indexes. The contribution array is segmented based on node indexes.

(iii) Apply a parallel segmented prefix sum scan [40] to integrate all node values.

All of these steps are implemented efficiently in parallel on the GPU. The first step is simply a pointwise computation. For the second step, we apply the fast parallel sorting [41]. The third step is performed using the optimal segmented scan function in the CUDA Performance Processing library (CUDPP) [40]. The sorting scheme on CUDA is a magnitude faster than an optimal multithreaded, multicore implementation on CPUs [29]. While this scheme is quite efficient and is the only solution on CUDA 1.0 devices, its performance largely depends on implementations of two essential functions: the parallel sorting and the segmented scan. Also the memory requirement of the method is proportional to the number of shooting points (which can be as large as the grid size) and the size of the neighbor (which is eight for 3D implementation). The memory usage become even worse as fast parallel sorting based on radix sorting that could not perform in-place but out-of-place sorting so the method requires another copy of the contribution array. In many circumstances, we found a better solution both in terms of performance and memory usage based on atomic operations supported on the CUDA 1.1 and later devices.

The *atomic splatting* scheme resolves the shared-resource fighting problem using atomic operations. While atomic floating point operations are currently not supported, it is possible to simulate this operation based on a fixed-point presentation. In particular, instead of accumulating the floating point buffer, we explicitly convert floating point values to integer representations through a scale. This allows the accumulation to be performed on integer buffers.

The parallel splatting accumulation is implemented by assigning each geometrical descriptor a GPU thread, which computes the contribution to the neighbor grid points based on its current value and distances to the neighbor grids. These floating point contribution values are then converted to integer presentation through a scale number, which is normally chosen as a power of two (we use 2^{20}, in practice) so that a fast shifting function is sufficient to perform the scale. The atomic integer adding operator allows values to be accumulated atomically at each grid point concurrently from thousand of threads. In our implementation, the contribution computations—upscale and the integer accumulation steps—are merged to one processing kernel to eliminate (1) an extra contribution buffer, (2) extra memory bandwidth usage to store, reload, and rescale the contribution buffer from the global memory, and (3) the call overheads of the three different GPU processing kernel. The accumulation result is then converted back to floating value by the division to the same scale value.

We further amortize the performance on later generation of GPU devices using the atomic shared-memory operations, which are a magnitude faster than operations on GPU global memory. We exploit the fact that in diffeomorphic registration the velocity field is often smooth and show large coherence between neighbors, so it is likely that two close points will share the same neighbors. Thus, it would be better to accumulate the values of the shared neighbors in the shared memory instead of the global memory. We assign each block of threads a close set of splatting points and maintain a shared memory accumulation buffer between threads of the same block. The accumulation results on the shared memory are then atomically added to the accumulation buffer on the global memory. This approach exploits the fast atomic functions on the shared memory and at the same time reduces the number of global atomic operations. This optimization is especially effective on a dense velocity

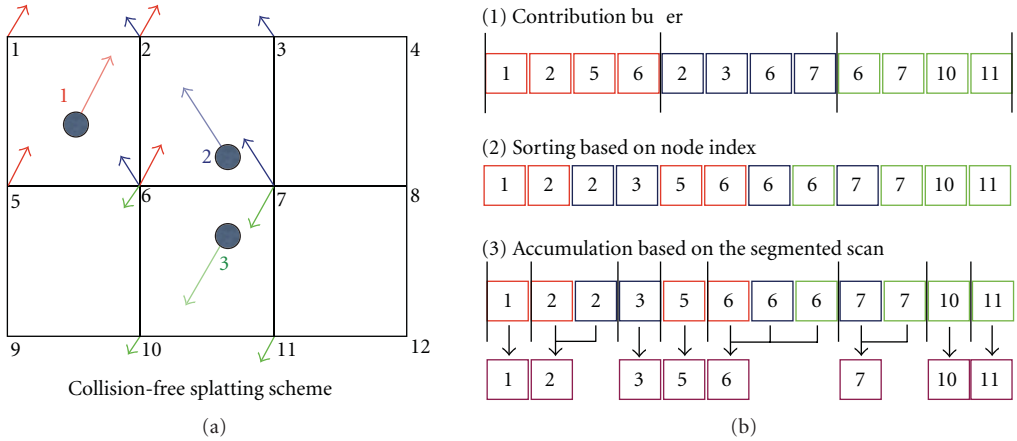


Figure 6: Collision-free splatting implementation using fast parallel sorting. The method is based on ordering the node contribution ID to resolve resource conflicts which allows a parallel efficient integration based on an optimal parallel prefix scan implementation.

field, which shows significant coherency between neighbor points.

4.2.3. Interpolation. Even though the probabilistic and geometric descriptors are represented by independent data structures on separate domains, they are, in fact, different representatives of the same anatomical subject that is updated during ODE integration under the influence of the time-dependent velocity field along a registration evolution path. While the computation occurs on the regular grid, interpolation is necessary to maintain the consistency of multicompartment anatomies as they undergo deformation. Given a deformation h, we update probabilistic images using backward mapping and geometries using forward mapping (3).

A computationally efficient version of ODE integration is the recursive equation that computes the deformation at time t based on the deformation at the time $t - 1$. That is, $h_t = h_{t-1}(x + v(t - 1))$. This computation is done by a reverse mapping operator (Figure 7), which assigns each destination grid point a value interpolated from the source volume grid's neighbor points. The reason for using a reverse mapping operator instead of a forward mapping one is to avoid missing data values at the grid points that makes computation of forward mappings intractable. A reverse mapping requires the maintenance of reverse velocity fields. The update of geometric descriptors is based on a forward vector field derived by inverting direction of the reverse velocity field. Algorithmically, the probabilistic and geometric descriptors are updated in opposite directions. The updating process of geometric descriptors is illustrated in Figure 8.

While the selection of interpolation strategies such as 3D linear interpolation, cubic interpolation, high-order interpolation depends on the quality requirement of the registration, the updating process of both probabilistic and geometric descriptor needs to share the same interpolation strategy so that they are consistent with one another. In practice, 3D linear interpolation is the most popular technique because it is computationally simple and efficient and it can produce satisfactory results especially with large kernel width for currents norm. On GPUs, this interpolation process is fully hardware accelerated with 3D texture volume support from CUDA 2.0 APIs. Another optimization is based on the texture cache that helps improve the lookup time from the source volume due to large coherency in the diffeomorphic deformation fields.

4.3. Other Performance Optimizations. Besides an optimized, parallel implementation for particle mesh computation, we further improve the performance with parallel surface normal and multiscale computation on GPUs. These optimizations keep the entire processing flow on GPUs, eliminating the need to transfer the data back and forth between CPU memory and GPU memory which is the main bottleneck for many GPU applications.

4.3.1. Parallel Surface Normal Computation on GPUs. While the geometrical descriptor involved in our registration framework was defined as a surface element (a triangle) with all property values on its vertices, the computation was defined at the centroid following its normal direction and weighted by the size of the surface element (11). This computation requires the computation of a weighted normal at the centroid of each surface element from the geometric descriptors. We perform this operation in parallel on the GPU by assigning each surface element a thread. We then employ the texture cache to load the geometrical data from global memory; while the neighbor triangle shared the same vertices, the loading values are highly likely in the cache and cost almost the same amount of time to access from the shared memory. We also store the three components of the normal in three separated arrays to allow coalesced access that gives better memory bandwidth efficiency.

4.3.2. Multiscale Computation on GPUs. Multiscale registration is an advanced registration technique to improve quality of the results by registering anatomies at different scale levels. The method also handles the local optimal matching of

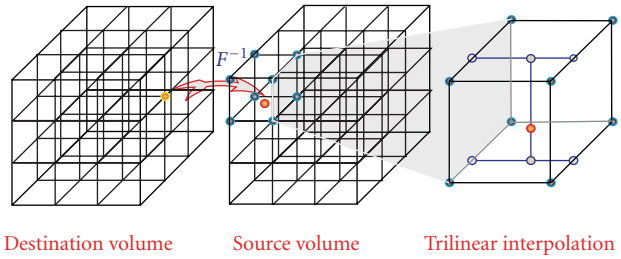


Figure 7: Reverse mapping based on 3D trilinear interpolation that eliminates the missing data of a forward mapping. The implementation on GPU exploits the hardware interpolation engine to achieve significant speedup.

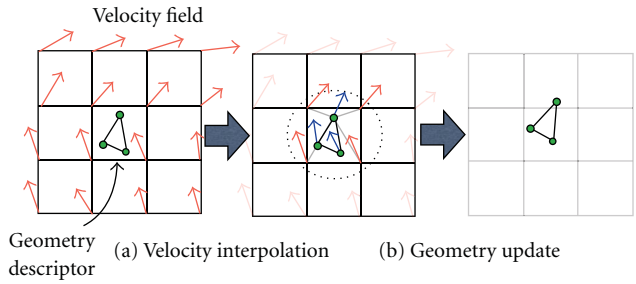


Figure 8: Geometries are updated through the interpolation from the velocity field. This step maintains the consistency between probabilistic and geometrical compartments of the mixture model.

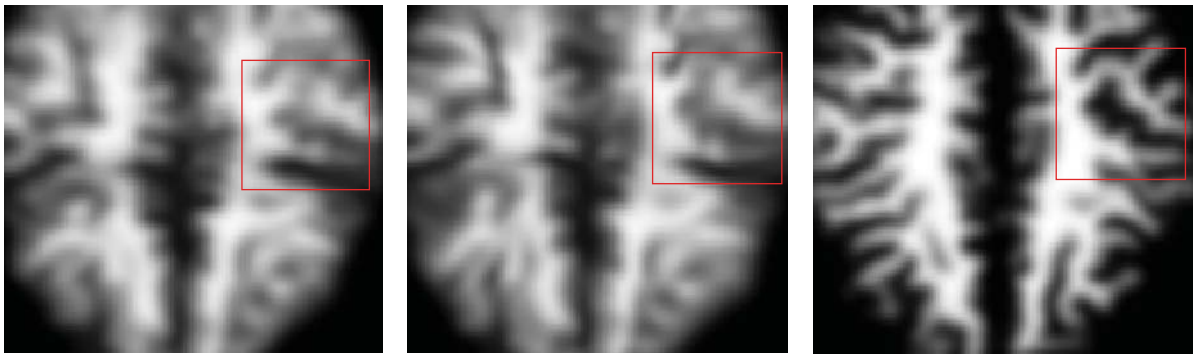
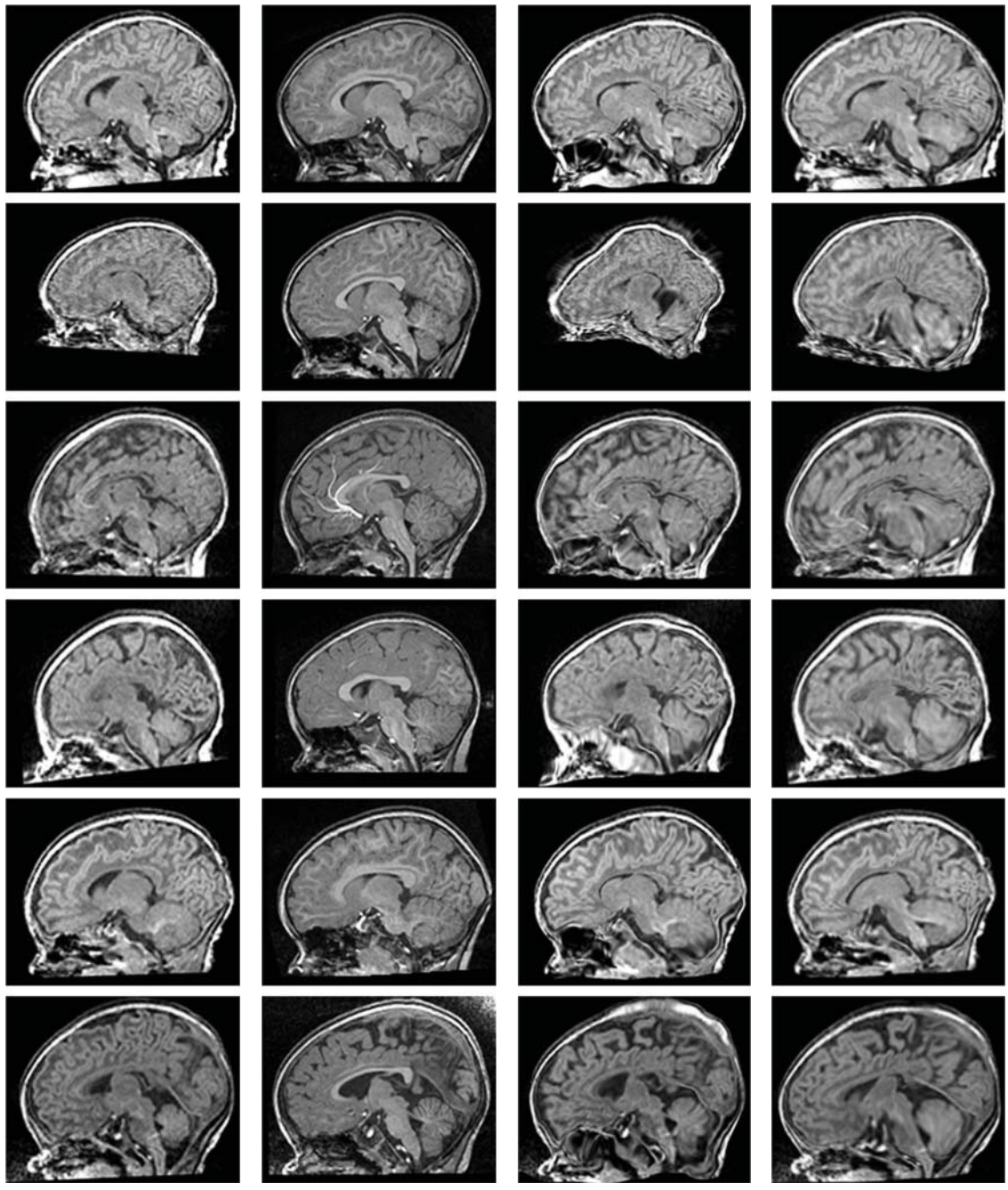


Figure 9: Multiscale registration using different sizes of computation kernels helps capture large- and small-scale changes in different levels and also increases the convergence rate of the algorithm.

gradient-descent optimization. In our registration framework, the primary purpose of doing multiscale computation is to capture both the large changes in the shape and also the small changes as the registration anatomy converged to the target. The method effectively handles the nonlinear, localized shape changes, as is shown in Figure 9. It also serves as an effective method to increase the convergence rate and reduces the running time significantly. The challenge of applying multiscale computation is that there is no mathematical foundation for exact multiscale computation on a regular grid. The level-of-detail techniques (LOD) are the only approximations that gives no guarantee on the quality. Here, we achieve the multiresolution effect through changing the size of a registration kernel, such that we use a larger kernel width and step size to mimic the effect of large-scale and smaller kernel width and step size to capture the details. Our method did not require resampling of the grids, so there are no additional quantization errors.

5. Results

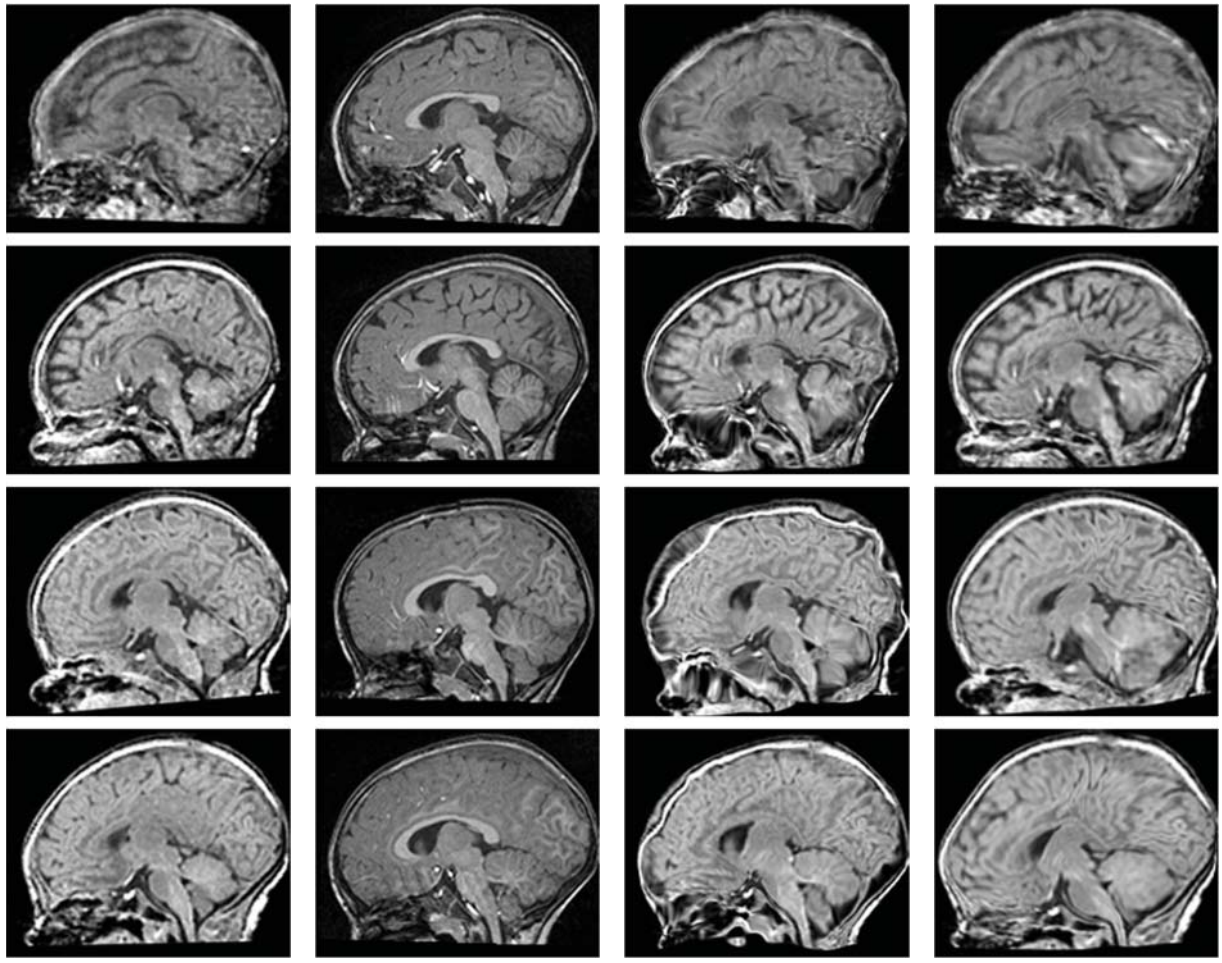
For evaluation, we used an AMD Phenom II X4 955 CPU commodity system, 6 GB DDR3 1333, with NVIDIA GT0260 GPU 896 MB. We quantify both aspects of the method: registration quality and performance. Runtime is measured in millisecond.



(a) (b) (c) (d)

Figure 10: Registration results of neonates mapped to 2-year olds. From left to right: (a) neonatal T1 image after affine registration, (b) reference T1 image at 2 years, followed by (c) neonatal T1 after deformable mutual information registration using B-splines, and (d) after combined probabilistic and geometric registration. From top to bottom: subjects 0012, 0102, 0106, 0121, 0130, and 0146. We note that the initial affine registration for subject 0102 (second row, second column) is incorrect; however our method managed to compensate and generate improved result compared to deformable mutual information registration.

5.1. Registration Quality. We have applied the registration method for mapping neonatal MRI scans to 2-year MRI scans of the same subjects in ten datasets. The datasets are taken from an ongoing longitudinal neuroimaging study with scans acquired at approximately two weeks, one year, and two years of age. Due to rapid early brain development, each longitudinal MR scan shows significant changes in brain size and in tissue properties. For comparison, we also applied the standard intensity-based deformable registration using mutual information (MI) metric and B-spline transformation proposed by Rueckert et al. [7], which has been applied for registering 1-year old and 2-year old infants [46].



(a) (b) (c) (d)

Figure 11: Registration results of neonates mapped to 2-year olds. From left to right: (a) neonatal T1 image after affine registration, (b) reference T1 image at 2 years, followed by (c) neonatal T1 after deformable mutual information registration using B-splines, and (d) after combined probabilistic and geometric registration. From top to bottom 0156, 0174, 0177, and 0180.

Both deformable registration methods are initialized using the same global affine transformation generated using the mutual information metric. The T1-weighted images before and after registration using the different approaches for the first three subjects are shown in Figures 10 and 11.

A quantitative study of the performance of the registration method is performed by measuring the overlap between the transformed segmentation maps of neonates to the segmentation maps of 2-year olds. Since we consider the segmentation maps at two years of age to be the standard, we use the following overlap metric:

$$\text{Overlap}(h \cdot S0, S2) = \frac{|h \cdot S0 \cap S2|}{|S2|}, \tag{16}$$

where $h \cdot S0$ is the transformed neonate segmentation map, $S2$ is the reference 2-year segmentation map, and $|\cdot|$ indicates the volume of a binary map. We note that this metric gives considerably lower values for deviation from $S2$ than the standard Dice coefficient. Table 1 shows the quantitative analysis for the brain parenchyma (a combination of white matter and grey matter) and cerebellum segmentation maps without registration, using standard MI registration, and our method. We use brain parenchyma since white matter and grey matter on their own are hard to distinguish in early developing brains. Registration using MI fails for parenchyma because it does not account for the two white matter distributions in neonates. Registration using both probabilistic and geometric descriptors provides better results and is generally more stable for the structures of interest. In particular, our method better preserves the shape of the cerebellum, which has weak intensity boundaries in regions where it touches the cerebrum and thus cannot be registered properly using only image-based information. Another significant challenge is that the cerebellum growth is distinctly different from the growth of neighboring structures. Using cerebellum boundary represented by currents, our method captures the growth better than MI registration.

Table 1: Overlap measures comparing the registered segmentation maps against the reference segmentation maps for the parenchyma and cerebellum structure, obtained through without deformation (None), deformable mutual information registration (MI), and our proposed method (P + G).

	Subject	0012	0102	0106	0121	0130	0146	0156	0174	0177	0180
Parenchyma	None	0.829	0.545	0.813	0.833	0.921	0.750	0.818	0.837	0.782	0.707
	MI	0.799	0.449	0.754	0.777	0.902	0.708	0.780	0.832	0.774	0.687
	P + G	0.903	0.883	0.884	0.868	0.881	0.860	0.875	0.879	0.913	0.874
Cerebellum	None	0.573	0.263	0.506	0.506	0.638	0.555	0.535	0.503	0.526	0.593
	MI	0.755	0.212	0.588	0.515	0.732	0.820	0.713	0.569	0.631	0.777
	P + G	0.881	0.821	0.875	0.878	0.858	0.899	0.907	0.885	0.896	0.892

Table 2: Runtime comparison, in milliseconds, of different splatting implementations on volume sized $144 \times 192 \times 160$ and $160 \times 224 \times 160$ using collision-free sorting approach, atomic operation with fixed point presentation, atomic operation on the shared memory and CPU reference.

Size	Method	CPU	Sorting	Atomic	Atomic shared
$144 \times 192 \times 160$	Random	826	105	29	30
	Diffeomorphic	331	110	105	14
	Singular	224	105	40	41
$160 \times 224 \times 160$	Random	1435	215	75	76
	Diffeomorphic	775	224	152	21
	Singular	347	215	144	144

Table 3: Runtime comparison, in milliseconds, of different 3D interpolation implementations for reverse mapping operator without memory caching (GPU global), with linear texture cache (1D linear) and hardware accelerated interpolation using 3D texture. The GPU-accelerated implementation is about 40 times faster than CPU reference and gives identical results.

Method	CPU	GPU global	1D linear	3D texture
$256 \times 256 \times 256$	777	30	24	19
$160 \times 224 \times 160$	209	10.4	7.3	6.8
$144 \times 192 \times 160$	173	6.8	4.8	5.4
$160 \times 160 \times 160$	149	6.6	5.0	5.2

5.2. Performance. We quantify the performance with two critical steps in particle mesh approach: the splatting and the interpolation. We measured the performance with typical volume sizes.

Splatting. The splatting performance varies largely depending on the regularity of the deformation fields due to memory collision problem. Here we measured with three types of deformation fields: a random deformation, which maps points randomly over the whole volume, a diffeomorphic deformation, the typical type of deformation from the registration of brain images that we use in our framework, and a singular deformation, which collapses to a point in the volume. Table 2 shows the runtime comparison in milliseconds of different splatting implementations mentioned in Section 4.2.2: CPU reference, collision-free sorting approach, atomic fixed-point operation, and atomic operation with shared memory.

The result shows that the performance gain of GPU approaches varies depending on the regularity of the deformation field inputs. The singular deformation has the lowest performance gain because most of the value accumulated to a small point neighbor hence parallel accumulation is greatly limited. Though having better performance gain, the random deformation spreads out in the whole volume that leads to ineffective caching (both in GPUs and CPUs). Fortunately, our atomic optimization with shared memory achieved the best performance gain with diffeomorphic deformation which we used in practice. The main reason is that the diffeomorphic deformation shows large coherence between neighbor points that allows more effective caching through GPU shared memory. The collision-free approach based on sorting shows stable performance since it is independent from the memory collision of other approaches.

Interpolation. Table 3 shows the runtime comparison in milliseconds of different 3D interpolation implementations: CPU reference, simple approach (GPU global memory), linear 1D texture, and 3D texture.

The interpolation runtime shows that reverse mapping using the accelerated hardware achieves the best performance and is about 38x faster than CPU reference implementation on the evaluation hardware. However, this method suffers from lower floating point accuracy. To not further introduce more errors to the approximation, we apply the 1D-linear texture-cache implementation instead which is as fast as the accelerated hardware but retains the floating point precision. The method produces results equivalent to the CPU reference.

Table 4: Time elapsed, in minutes, for registration using deformable mutual information (MI) on the CPU (AMD Phenom II X4 955, 6 GB DDR3 1333) and our proposed approach (P + G) on the GPU (NVIDIA GTX 260, 896 MB) with 1000 iterations of gradient descent.

Subject	0012	0102	0106	0121	0130	0146	0156	0174	0177	0180
MI on CPU	92	63	103	92	101	112	106	99	91	96
P + G on GPU	9	8	8	8	8	7	9	8	7	7

Overall Performance. We have also compared the performance between our method and the standard MI registration. Registration using our approach on the GPU takes 8 minutes on average, while registration on the CPU using mutual information metric and B-spline transformation takes 100 minutes on average. Detailed time measures are listed in Table 4.

Overall, computing the currents norm and its gradient between a surface with 160535 triangular faces and another with 127043 faces takes approximately 504 seconds on CPU, while it takes 0.33 seconds with our GPU implementation. The speed gain is in order of three magnitudes over the equivalent CPU implementation using particle mesh, while the computing time for the exact norm on CPU is difficult to measure since it takes significantly longer. The proposed algorithm typically converges in 1000 iterations, so on average it takes less than eight minutes to register two anatomies. This allows us to perform parameter exploration and real-time analysis on a single desktop with commodity GPU hardware.

6. Conclusions

We have proposed a registration framework that makes use of the probabilistic and geometric structures of anatomies embedded in the images. This allows us to enforce matching of important anatomical features represented as regional class posteriors and tissue boundaries. Our framework allows us to register images with different contrast properties by using equivalent anatomical representations, and we have demonstrated results for registering brain MRIs with different white matter appearances at early stages of growth. The overlap validation measures in Table 1 show that geometric constraints, particularly for the cerebellum, are crucial for registering structures undergoing significant growth changes.

In the future, we plan to apply this framework in early neurodevelopmental studies for analyzing the effects of neurological disorders such as autism and fragile X syndrome. The proposed registration framework is generic and independent of the application domain; it can thus be applied to any registration where one encounters large-scale deformation and different appearance patterns. We also want to incorporate other submanifolds representations and their computation such as point sets ($\mathcal{M}(0)$) and curves ($\mathcal{M}(1)$). Such additional representations are potentially critical in clinical applications involving anatomical landmark points (e.g., anterior commissure and posterior commissure) as well as curve structures (e.g., blood vessels, sulcal lines, white matter fiber tracts). All these computations can be done efficiently and entirely on GPUs and potentially will improve the results by guiding the registration process to preserve critical geometries. The efficiency of the GPU method also provides an opportunity to apply the algorithm for high-quality atlas formation using our framework on a GPU cluster, which gives us the ability to perform statistical tests that are previously impossible due to excessive time requirements.

Acknowledgments

This work is supported by NIH Grants 5R01EB007688 and Conte Center MH064065, UCSF Grant P41 RR023953, and NSF Grant CNS-0751152.

References

[1] J. Glaunes, A. Trouvé, and L. Younes, "Diffeomorphic matching of distributions: a new approach for unlabelled point-sets and sub-manifolds matching," in *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '04)*, pp. 712–718, July 2004.

[2] P. Lorenzen, B. Davis, and S. Joshi, "Unbiased atlas formation via large deformations metric mapping," in *the 8th International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI '05)*, vol. 3750 of *Lecture Notes in Computer Science*, pp. 411–418, October 2005.

[3] M. I. Miller and L. Younes, "Group actions, homeomorphisms, and matching: a general framework," *International Journal of Computer Vision*, vol. 41, no. 1-2, pp. 61–84, 2001.

[4] R. C. Knickmeyer, S. Gouttard, C. Kang et al., "A structural MRI study of human brain development from birth to 2 years," *Journal of Neuroscience*, vol. 28, no. 47, pp. 12176–12182, 2008.

[5] M. Datar, J. Cates, P. T. Fletcher, S. Gouttard, G. Gerig, and R. Whitaker, "Particle based shape regression of open surfaces with applications to developmental neuroimaging," in *the 12th International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI '09)*, vol. 5762 of *Lecture Notes in Computer Science*, pp. 167–174, 2009.

[6] H. Xue, L. Srinivasan, S. Jiang et al., "Longitudinal cortical registration for developing neonates," in *the 10th International Conference on Medical Imaging and Computer-Assisted Intervention (MICCAI '07)*, vol. 4792 of *Lecture Notes in Computer Science*, pp. 127–135, October 2007.

[7] D. Rueckert, L. I. Sonoda, C. Hayes, D. L. G. Hill, M. O. Leach, and D. J. Hawkes, "Nonrigid registration using free-form deformations: application to breast mr images," *IEEE Transactions on Medical Imaging*, vol. 18, no. 8, pp. 712–721, 1999.

[8] C. E. Scheidegger, J. M. Schreiner, B. Duffy, H. Carr, and C. T. Silva, "Revisiting histograms and isosurface statistics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 6, pp. 1659–1666, 2008.

[9] G. E. Christensen, M. I. Miller, M. W. Vannier, and U. Grenander, "Individualizing neuroanatomical atlases using a massively parallel computer," *Computer*, vol. 29, no. 1, pp. 32–38, 1996.

[10] A. Eklund, M. Andersson, and H. Knutsson, "Phase based volume registration using CUDA," in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '10)*, pp. 658–661, March 2010.

[11] P. Micikevicius, "3D finite difference computation on GPUs using CUDA," in *the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2 '09)*, p. 79, March 2009.

[12] M. Roberts, M. C. Sousa, and J. R. Mitchell, "A work-efficient GPU algorithm for level set segmentation," in *the 37th International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '10)*, July 2010.

[13] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, and W. M. W. Hwu, "GPU acceleration of cutoff pair potentials for molecular modeling applications," in *the 5th Conference on Computing Frontiers (CF '08)*, pp. 273–282, May 2008.

[14] L. Ha, J. Kruger, S. Joshi, and C. T. Silva, *Multiscale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs*, vol. 1, Elsevier, New York, NY, USA, 2011.

[15] B. Avants and J. C. Gee, "Geodesic estimation for large deformation anatomical shape averaging and interpolation," *NeuroImage*, vol. 23, no. 1, pp. S139–S150, 2004.

[16] T. Rohlfing, D. B. Russakoff, and C. R. Maurer, "Expectation maximization strategies for multi-atlas multi-label segmentation," in *the 18th International Conference on Information Processing in Medical Imaging*, vol. 2732 of *Lecture Notes in Computer Science*, pp. 210–221, 2003.

[17] P. M. Thompson and A. W. Toga, "A framework for computational anatomy," *Computing and Visualization in Science*, vol. 5, no. 1, pp. 13–34, 2002.

[18] U. Grenander and M. I. Miller, "Computational anatomy: an emerging discipline," *Quarterly of Applied Mathematics*, vol. 56, no. 4, pp. 617–694, 1998.

[19] F. L. Bookstein, *Morphometric Tools for Landmark Data: Geometry and Biology*, Cambridge University Press, 1991.

[20] M. I. Miller, G. E. Christensen, Y. Amit, and U. Grenander, "Mathematical textbook of deformable neuroanatomies," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 90, no. 24, pp. 11944–11948, 1993.

[21] G. E. Christensen, R. D. Rabbitt, and M. I. Miller, "Deformable templates using large deformation kinematics," *IEEE Transactions on Image Processing*, vol. 5, no. 10, pp. 1435–1447, 1996.

[22] M. F. Beg, M. I. Miller, A. Trouvé, and L. Younes, "Computing large deformation metric mappings via geodesic flows of diffeomorphisms," *International Journal of Computer Vision*, vol. 61, no. 2, pp. 139–157, 2005.

[23] V. Camion and L. Younes, "Geodesic interpolating splines," in *Proceedings of the 3rd International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition (EMMCVPR '01)*, pp. 513–527, London, UK, 2001.

[24] S. C. Joshi and M. I. Miller, "Landmark matching via large deformation diffeomorphisms," *IEEE Transactions on Image Processing*, vol. 9, no. 8, pp. 1357–1370, 2000.

[25] M. Vaillant and J. Glaunès, "Surface matching via currents," in *the 19th International Conference on Information Processing in Medical Imaging (IPMI '05)*, pp. 381–392, July 2005.

[26] J. Glaunès, A. Qiu, M. I. Miller, and L. Younes, "Large deformation diffeomorphic metric curve mapping," *International Journal of Computer Vision*, vol. 80, no. 3, pp. 317–336, 2008.

[27] A. W. Toga and P. Thompson, "Brain warping," in *Brain Warping*, pp. 1–26, Academic Press, New York, NY, USA, 1999.

[28] J. A. Glaunes and S. Joshi, "Template estimation from unlabeled point set data and surfaces for computational anatomy," in *the International Workshop on Mathematical Foundations of Computational Anatomy*, 2006.

[29] B. C. Davis, E. Bullitt, P. T. Fletcher, and S. Joshi, "Population shape regression from random design data," in *the 11th IEEE International Conference on Computer Vision (ICCV '07)*, October 2007.

[30] S. Durrleman, X. Pennec, A. Trouvé, P. Thompson, and N. Ayache, "Inferring brain variability from diffeomorphic deformations of currents: an integrative approach," *Medical Image Analysis*, vol. 12, no. 5, pp. 626–637, 2008.

[31] K. Van Leemput, F. Maes, D. Vandermeulen, and P. Suetens, "Automated model-based tissue classification of MR images of the brain," *IEEE Transactions on Medical Imaging*, vol. 18, no. 10, pp. 897–908, 1999.

[32] S. K. Warfield, M. Kaus, F. A. Jolesz, and R. Kikinis, "Adaptive, template moderated, spatially varying statistical classification," *Medical Image Analysis*, vol. 4, no. 1, pp. 43–55, 2000.

[33] M. Prastawa, J. H. Gilmore, W. Lin, and G. Gerig, "Automatic segmentation of MR images of the developing newborn brain," *Medical Image Analysis*, vol. 9, no. 5, pp. 457–466, 2005.

[34] W. E. Lorensen and H. E. Cline, "Marching cubes: a high resolution 3D surface construction algorithm," *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 163–169, 1987.

[35] Z. Zhang, "Iterative point matching for registration of free-form curves and surfaces," *International Journal of Computer Vision*, vol. 13, no. 2, pp. 119–152, 1994.

[36] H. A. El Munim and A. A. Farag, "Shape representation and registration using vector distance functions," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '07)*, June 2007.

[37] K. M. Pohl, J. Fisher, S. Bouix et al., "Using the logarithm of odds to define a vector space on probabilistic atlases," *Medical Image Analysis*, vol. 11, no. 5, pp. 465–477, 2007.

[38] A. Tsai, A. Yezzi, W. Wells et al., "A shape-based approach to the segmentation of medical imagery using level sets," *IEEE Transactions on Medical Imaging*, vol. 22, no. 2, pp. 137–154, 2003.

[39] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, Article ID 4490127, pp. 879–899, 2008.

[40] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, *CUDPP: CUDA Data Parallel Primitives Library*, 2007.

[41] D. G. Merrill and A. S. Grimshaw, "Revisiting sorting for GPGPU stream architectures," in *the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*, pp. 545–546, September 2010.

[42] S. Durrleman, X. Pennec, A. Trouvé, and N. Ayache, "Sparse approximation of currents for statistics on curves and surfaces," in *the 11th International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI '08)*, vol. 5242 of *Lecture Notes in Computer Science*, pp. 390–398, September 2008.

[43] L. Greengard and J. Strain, "The fast gauss transform," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 1, pp. 79–94, 1991.

[44] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, Taylor and Francis, 1989.

[45] A. Trouvé and L. Younes, "Metamorphoses through lie group action," *Foundations of Computational Mathematics*, vol. 5, no. 2, pp. 173–198, 2005.

[46] P. Aljabar, K. K. Bhatia, M. Murgasova et al., "Assessment of brain growth in early childhood using deformation-based morphometry," *NeuroImage*, vol. 39, no. 1, pp. 348–358, 2008.

Hindawi Publishing Corporation
International Journal of Biomedical Imaging
Volume 2011, Article ID 640208, 12 pages
doi:10.1155/2011/640208

Research Article

Heterogeneous Computing for Vertebra Detection and Segmentation in X-Ray Images

Fabian Lecron, Sidi Ahmed Mahmoudi, Mohammed Benjelloun, Saïd Mahmoudi, and Pierre Manneback

Computer Science Department, Faculty of Engineering, University of Mons, Place du Parc, 20 7000 Mons, Belgium

Correspondence should be addressed to Fabian Lecron, fabian.lecron@umons.ac.be and Sidi Ahmed Mahmoudi, sidi.mahmoudi@umons.ac.be

Received 8 March 2011; Accepted 3 June 2011

Academic Editor: Yasser M. Kadah

The context of this work is related to the vertebra segmentation. The method we propose is based on the active shape model (ASM). An original approach taking advantage of the edge polygonal approximation was developed to locate the vertebra positions in a X-ray image. Despite the fact that segmentation results show good efficiency, the time is a key variable that has always to be optimized in a medical context. Therefore, we present how vertebra extraction can efficiently be performed in exploiting the full computing power of parallel (GPU) and heterogeneous (multi-CPU/multi-GPU) architectures. We propose a parallel hybrid implementation of the most intensive steps enabling to boost performance. Experimentations have been conducted using a set of high-resolution X-ray medical images, showing a global speedup ranging from 3 to 22, by comparison with the CPU implementation. Data transfer times between CPU and GPU memories were included in the execution times of our proposed implementation.

1. Introduction

The general context of the present work is the cervical vertebra mobility analysis on X-ray images. The objective is to be able to automatically measure the vertebral angular movements. The issue is to determine the angles between adjacent vertebræ on radiographs characterized by 3 patient's positions: flexion, neutral, and extension.

In [1], Puglisi et al. describe a protocol for the analysis of cervical vertebra mobility applied to hospital diagnosis or to scientific research. They show that for the mobility analysis, the vertebra contour needs to be defined by a human operator. This operation can be performed on the original or the digitized radiograph. One contribution of this paper is to develop automated methods to provide such data.

One way to extract quantitative data is to segment the vertebræ by digital image processing. Nowadays, regular radiography of the spine is the cheapest and the fastest way for a physician to detect vertebral abnormalities. Furthermore, as far as the patient is concerned, this procedure is sure and noninvasive. Despite these advantages, the segmentation of X-ray images is very challenging due to their nature. They are characterized by very poor contrast, blended contours, and the human body complexity is responsible for the fact that some parts of the image could be partially or completely hidden by other organs. In a medical context, a key variable has to be taken into account: the time. It is crucial to develop efficient applications with a reduced execution time, especially in the case of urgent diagnosis. To do so, one can imagine to use a parallel-based architecture such as cluster, grid computing, graphics processing units (GPUs).

The GPUs represent an efficient solution to solve this problem. However, such a solution does not exploit the CPU multiple computing units (cores) present in the majority of computers. Moreover, the solution based on GPU is seriously hampered by the high costs of data transfer between CPU and GPU memories. To limit these constraints, we propose a parallel hybrid implementation which allows exploiting effectively the full computing power of heterogeneous architectures. Notice that heterogeneous architectures dispose of both multiple CPU and multiple GPU cores. The proposed implementation is applied on the most intensive step of

vertebra segmentation method. Indeed, we develop a parallel hybrid implementation of the recursive contour extraction technique using Canny's criteria [2]. Our choice to parallelize this method is due to its noise robustness and its reduced number of operations. These factors allow applying the application on large sets of medical images and enable to have more precise results for vertebra extraction. Our work is especially dedicated to the use of large images databases. Therefore, our framework could be used in a medical context given the growing number of patients. Another application could be associated to the search and the navigation in large images and videos databases, such as in [3].

The remainder of the paper is organized as follows: related works are described in Section 2. Section 3 presents the CPU implementation of the proposed method based on active shape model. Section 4 discusses the use of GPU for image processing algorithms, while Section 5 is devoted to the parallel hybrid implementation of our approach, exploiting effectively the full computing power of heterogeneous architectures. Section 6 presents the obtained results of vertebra extraction using a data set of medical images and compares the performance between CPU, GPU and hybrid implementations. Finally, Section 7 concludes and proposes further work.

2. Related Work

One can find two kinds of related work for which vertebra segmentation and optimal edge detection in medical images are the fundamental processing steps: the first one is related to sequential solutions for vertebra extraction using CPUs, and the second is related to the use of GPU to accelerate image processing algorithms, which can be exploited for medical applications.

2.1. Vertebra Segmentation on CPU. If we study the segmentation approaches described in the literature, we can observe that their effectiveness depends on the related medical imagery modality. One can distinguish 3 types of modality: the conventional radiography (X-ray), the computed tomography (CT) and the magnetic resonance (MR).

With regard to the MR images, a watershed algorithm has been used in [4] to segment and reconstruct intervertebral disks. The idea is to provide preoperative data with an image-guided surgery system. The method uses a combination of statistical and spectral texture features to discriminate closed regions representing intervertebral disks. Recently, Huang et al. have used a learning-based approach applied to the vertebra detection and segmentation on MR images [5]. To this end, features such as Harr wavelet are extracted on images to train an AdaBoost learning algorithm. Finally, a normalized graph cut algorithm is used to segment the exact contour of the vertebræ. A similar approach has also been proposed for the vertebra segmentation in the context of CT images. In [6], lumbar vertebræ are segmented by the minimization of the graph cut associated to a CT image.

Still in this context, the active contour algorithm which deforms and moves a contour submitted to internal and external energies is applied in [7]. In this work, Klinder et al. provides a framework dedicated to the detection, identification, and segmentation of CT images for the computer-assisted surgery. Concerning the segmentation part, they use a constrained deformable model defined in [8]. In the same idea, the level set method, which makes an interface evolve in the image, has also been dedicated to the vertebra segmentation in [9, 10]. The main drawback of these methods remains the strong influence of an initialization close to the target.

To deal with X-ray images, methods only based on the image information are not adapted. The efficient methods for MR or CT images are not suitable for radiographs because of the blended contours. An exact segmentation needs additional details about the object of interest. For this reason, a template matching algorithm combined with a polar signature system has been proposed in [11]. Other model-based methods such as active shape model [12] and active appearance model [13] showed their effectiveness. Basically, an active shape model is a statistical model generated from a set of training samples. A mean shape model is computed and placed near the vertebræ of interest on the radiograph. ASM search applies deformations on this mean shape so that it corresponds to the real vertebra contour. An active appearance model is based on the same principle but introduces a model of the underlying distribution of intensity around the landmarks. In this paper, since we do not need the information about the texture, we decided to use active shape model to characterize and segment the vertebræ. ASM and AAM have been, respectively, used in [14–16] and [17–20] for the vertebra segmentation. However, the models used are global ones, that is, defined by several vertebræ. The interest of that model is to provide information about the curvature and the dependence between two vertebræ. Nevertheless, in the context of the vertebral mobility analysis, the global models cannot explain all the curvature variability since 3 particular patient's positions are studied. The only way to achieve the segmentation is to use a local vertebra model. However, in order to ensure an exact contour extraction, we need to precisely initialize the segmentation step by placing mean shape very close to the vertebræ of interest. In the literature, the generalized Hough transform (GHT) is often used for that matter. In [21], the authors try to take advantage of the GHT on radiographs in a fully automatic way, but they present a segmentation rate equal to 47% for lumbar vertebræ without providing information about the detection rate. Very recently, Dong and Zheng have proposed a method combining GHT and the minimal intervention of a user with only 2 clicks in the image [22].

2.2. GPU for Image Processing. Many image processing and rendering algorithms are known by their high consumption of both computing power and memory. Beyond of image rendering, most of image processing algorithms contain phases which consist of similar calculations between image pixels. These facts make these algorithms prime candidates

for acceleration on GPU by exploiting processing units in parallel. In this category, Yang et al. implemented several classic image processing algorithms on GPU with CUDA [23]. OpenVIDIA project [24] has implemented different computer vision algorithms running on graphics hardware such as single or multiple graphics processing units, using OpenGL [25], Cg [26], and CUDA [27]. Luo and Duraiswami proposed a GPU implementation [28] of the Canny edge detector [29]. There are also some GPU works in medical imaging for new volumetric rendering algorithms [30, 31] and magnetic resonance (MR) image reconstruction on GPU [32]. Otherwise, there are different works for the exploitation of heterogeneous architectures of multicores and GPUs. Ayguadé et al. proposed a flexible programming model for multicores [33]. StarPU [34] provides a unified runtime system for heterogeneous multicore architectures (CPUs and GPUs), enabling to develop effective scheduling strategies.

Actually, our contribution is to propose firstly an original automated approach to detect the vertebra location in a radiograph which will be used for the initialization of the segmentation phase. Next, we develop a model-based segmentation procedure especially adapted to the vertebral mobility study. We also contribute with improving performance of vertebra segmentation in X-ray medical images, by implementing the most intensive step of the proposed approach on heterogeneous architectures composed of both CPUs and GPUs. Indeed, we propose a parallel hybrid implementation of edge detection step using Deriche-Canny method [2] that enables a better noise removing and requires a less number of operations than Canny method. This hybrid implementation allows an efficient exploitation of the full computing power of heterogeneous architectures (multi-CPU/multi-GPU) and enables to more improve performance than the GPU implementation described in [35].

3. General Framework

In this paper, we propose an original approach for the vertebra segmentation in the context of the vertebral mobility analysis. Our method is actually based on the active shape model. Global statistical models generally used in the literature are not able to explain the spine curvature variability induced by the 3 positions: extension, neutral, and flexion. Therefore, we decided to use a local vertebra model implying an exact initialization of the ASM-based segmentation, that is, placing the mean shape close to the vertebræ of interest. For that matter, we need to locate the position of all the vertebral bodies. To this end, vertebra features are detected according to an original procedure. Actually, the anterior corners of each vertebra are located in the radiograph by an approach based on the edge polygonal approximation. Once we have extracted the vertebra position, we can start the segmentation procedure.

3.1. Learning. The learning phase starts with the creation of a sample of images. In our case, we use X-ray radiographs focused on the cervical vertebræ C3 to C7 (see Figure 2). Actually, each item of the sample has to be described by an information, that is, the coordinates of some landmarks located on the item contour. These points need to correspond on the different shapes in the sample. The resulting marked vertebræ are not directly exploitable. Every shape in the sample has particular position and orientation. Building a relevant statistical model requires to align the shapes. To this end, we use an alignment approach based on the Procrustes analysis and detailed in [36].

3.2. Modelization. As soon as all the vertebra shapes are aligned, they can be used to build the active shape model. To do so, the mean shape is first computed and then a group of other allowable shapes is derived by moving the landmarks in specific directions, obtained by a principal component analysis (PCA) of the data. We refer the reader to [12] for more detail about the modelization.

3.3. Initialization. In order to initialize the segmentation procedure, the mean shape has to be placed close to a vertebra of interest. In [37], we proposed an original method to locate points of interest in a radiograph. Here, we use part of this work but we also detect the vertebræ by their anterior corners. First, a user is asked to mark out 2 points in the image to determine a region of interest (ROI) by the higher anterior corner of the C3 vertebra and the lower anterior corner of the C7 vertebra. Then, all the vertebral bodies are detected with a process composed of 4 steps: a contrast-limited adaptive histogram equalization to improve the image contrast, an edge detector, an anterior corner detection, and finally the vertebra localization.

3.3.1. Contrast-Limited Adaptive Histogram Equalization. The X-ray images we deal with have very poor contrast. Before any further process, we need to improve the image quality. A simple histogram equalization has no impact on the radiographs. Therefore, we propose to use a specific method: the contrast-limited adaptive histogram equalization [38].

The principle is to divide the image in contextual regions. In each of them, the histogram is equalized. Nevertheless, this transformation induces visual boundaries around the contextual regions. To get rid of this effect, a bilinear interpolation scheme is used. Let A, B, C, and D be the centers of contextual regions (see Figure 1). For a pixel p_i with an intensity r, we can write

$$\begin{aligned} s = {} & (1-y)[(1-x)T_A(r) + xT_B(r)] \\ & + y[(1-x)T_C(r) + xT_D(r)] \end{aligned} \tag{1}$$

where T_k stands for the equalization transformation of the region k, and s for the new value of the pixel p_i.

Only applying this scheme is still dependent to the increase of the noise in the radiograph. One way to decrease it is to reduce the contrast improvement in the homogeneous areas. A contrast factor is then defined to limit the highest peaks in the equalized histogram. The pixels above this factor limit are uniformly redistributed in the histogram.

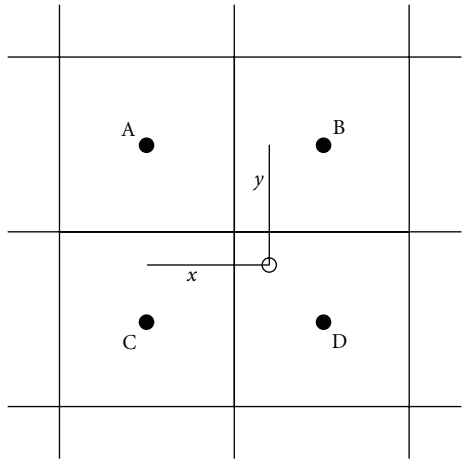


Figure 1: Image sample divided in 4 contextual regions.

3.3.2. Edge Detection. The Canny edge detector, introduced in [29] allows to detect edges in an image by taking advantage of the information given by the intensity gradient. Let I be this image. The first step is to reduce the noise by removing isolated pixels. To this aim, the image is convolved with the Gaussian filter defined by (2).

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2}. \quad (2)$$

Next, the Sobel operator is applied on the resulting image. Let A be this image. The operator is based on a couple of masks defined by the relation (3). With this information, the gradient of a pixel can be computed by

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * A \qquad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * A, \quad (3)$$

$$G = \sqrt{G_x^2 + G_y^2}, \quad (4)$$

Additional information about the gradient orientation is simply given by:

$$\theta = \arctan\left(\frac{G_y}{G_x}\right). \quad (5)$$

Once the gradient has been computed for every pixel, only maxima have to be retained. High gradient intensity stands for high probability of edge presence. Finally, the last phase makes a hysteresis binarization. High and low thresholds are defined in such a way that, for each pixel, if the gradient intensity is

(i) lower than the low threshold, the point is rejected,

(ii) greater than the high threshold, the point is part of the edge,

(iii) between the low and the high thresholds, the point is accepted only if it is connected to an already accepted point.

3.3.3. Corner Detection. In this work, we propose to locate the vertebræ by firstly detecting some features: the anterior vertebra corners. To this end, we present a method based on the geometrical definition of a corner; that is, a point is considered as a corner if it is located at the intersection of two segment lines. The idea is to perform an edge polygonal approximation. Usually, works about the polygonal approximation detect the dominant points in the image and build a polygonal approximation. Here, we do the opposite by using the polygonal approximation to detect features in the image.

The Canny algorithm provides the edges on the image but only acts on the pixel values. In order to carry out the polygonal approximation algorithm, we need to define the contours as sets of points. Therefore, a simple contour tracking approach has been developed.

The algorithm used in this paper is the one proposed by Douglas and Peucker in [39]. This approach is based on the principle that a polyline represented by n points can be reduced in a representation with 2 points if the distance between the segment line joining the extremities of the polyline and the farthest point from this line is lower than a given threshold. The first stage concerns the selection of the extremities E_1 and E_2 of the polyline. Let A be the farthest point from the segment line $\|E_1E_2\|$ and d the distance between the point A and $\|E_1E_2\|$. Three scenarios are considered:

(i) if $d \leq \epsilon$, all the points situated between E_1 and E_2 are removed,

(ii) if $d > \epsilon$, the algorithm is recursively applied on 2 new polylines: $\|E_1A\|$ and $\|AE_2\|$,

(iii) if there is no point between E_1 and E_2, the polyline is no longer reducible.

3.3.4. Vertebra Localization. Now that we have detailed how to detect corners in a general way, let us explain how to only detect the vertebra ones. Among all the corners detected by our approach based on the edge polygonal approximation, the ones describing a vertebra need to be distinguished.

The first stage of our procedure is to build a statistical model of the spine curvature in order to extract the mean shape. The landmarks of the model are the anterior vertebra corners. An illustration is given at the Figure 2. Notice that here, the goal is not to explain precisely the curvature but just to have a way to locate vertebra anterior corners. The next step brings a user to mark out the higher anterior corner of the C3 vertebra and the lower anterior corner of the C7 vertebra to define a ROI. Then, we perform an alignment between these two particular points and the mean shape of the spine curvature model. Finally, for each landmark, we search the closest corner detected by the approach based on the edge polygonal approximation. Note that a specific order has to be followed: from the top to the bottom of the image (the opposite could be considered). This order is crucial to avoid the algorithm swapping lower and higher corners of two successive vertebræ.

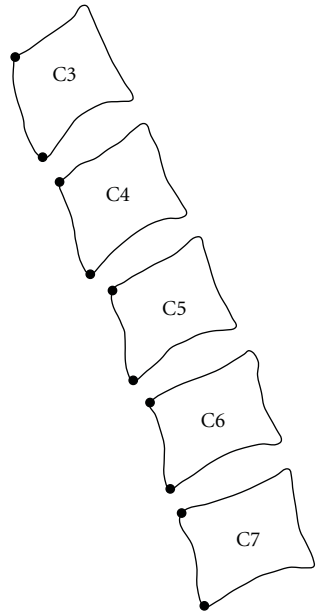


Figure 2: Landmarks for the spine curvature modelization.

3.4. Segmentation. The statistical model allowing to identify acceptable shapes of the object of interest is now defined. However, we still have to present how the search in the image is conducted during the segmentation. To this end, the grey level variation has to be locally evaluated around each landmark in the sample. Then, a mean profile of the texture (gradient intensity) can be deduced. After the initialization, a local analysis of the texture is carried out around each landmark of the initial shape. The goal is to find the best match with the mean profile previously determined. The distance used for the profile comparison is the Mahalanobis distance. This search implies that the landmarks are moved during the segmentation. The procedure is repeated until the convergence, that is, when the match between the current shape profile and the mean one is no more improved.

4. Image Processing on GPU

Image processing algorithms represent an excellent topic for acceleration on GPU, since the majority of these algorithms have sections which consist of a common computation over many pixels. This fact is due to the exploitation of the high number of GPU's computing units in parallel. As a result, we can say that graphics cards represent an efficient tool for boosting performance of image processing techniques. This section describes firstly the key factors of GPUs and the programming languages used to exploit their high power and secondly the proposed development scheme for image processing on GPU, based upon CUDA for parallel constructs and OpenGL for visualization.

4.1. GPU Programming. Graphics processing units (GPUs) have dramatically evolved during last years as shown in Figure 3. This evolution makes them a very high attractive

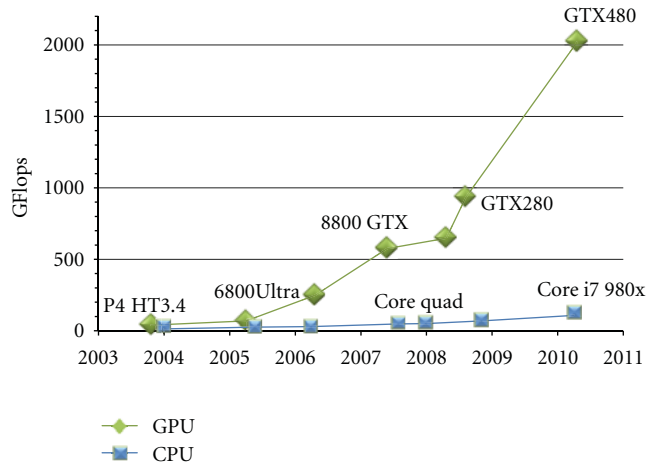


Figure 3: Computational Power: GPU versus CPU. Derived from [40].

hardware platform for general purpose computation. For a better exploitation of this high power, the GPUs memory bandwidth has also significantly increased. Furthermore, the advent of GPGPU (general purpose graphics processing unit) languages enabled exploiting GPU for more types of application and not only for image rendering and video games. In this context, NVIDIA launched the API CUDA (compute unified device architecture) [27], a programming approach which exploits the unified design of the most current graphics processing units from NVIDIA. Under CUDA, GPUs consist of many processor cores which can address directly to GPU memories. This fact allows a more flexible programming model. As a result, CUDA has rapidly gained acceptance in domains where GPUs are used to execute different intensive parallel applications.

4.2. Image Processing Model Based on CUDA and OpenGL. We propose in this paragraph a model for image processing on graphics processors, enabling to load, treat, and display images on GPU. This model is represented by a scheme development based upon CUDA for parallel constructs and OpenGL for visualization, which reduces data transfer between device and host memories. This scheme is based on four steps (Figure 4):

(i) copy input data,

(ii) threads allocation,

(iii) parallel processing with CUDA,

(iv) output results.

(1) *Copy Input Data:* The transfer of input data (images) from host (CPU) to device (GPU) memory enables to apply GPU treatments on the copied image.

(2) *Threads Allocation:* After loading the input data (images) on GPU memory, the threads number in the grid of GPU has to be selected so that each thread can perform its processing on one or a group of pixels. This allows threads to process in parallel on image

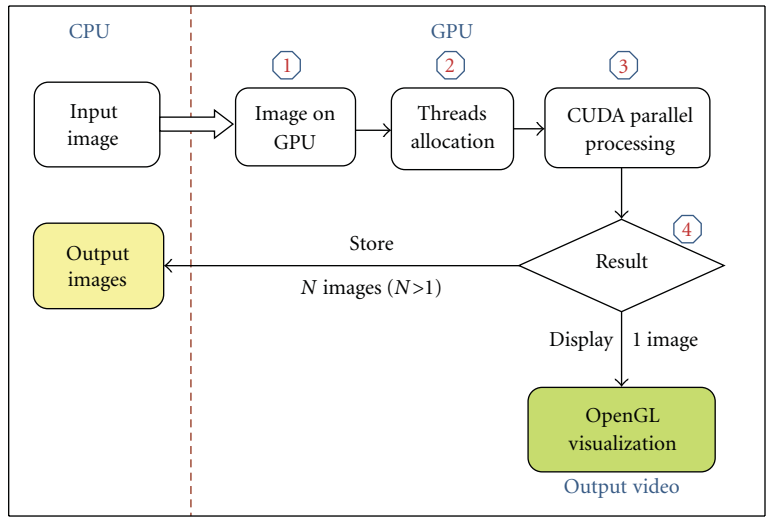


Figure 4: Image Processing on GPU based on CUDA and OpenGL.

pixels. We note that the selection of the number of threads depends on the number of pixels.

(3) *Parallel Processing with CUDA:* The CUDA functions (kernels) are executed N times using the N selected threads in the previous step.

(4) *Output Results:* After processing, results can be presented using two different scenarios.

(i) *OpenGL Visualization:* The visualization of output images using the graphics library OpenGL is fast, since it exploits buffers already existing on GPU. Indeed, the compatibility of OpenGL with CUDA enables to avoid more data transfer between host and device memories. This scenario is useful when parallel processing is applied on one image only since we cannot display many images using one video output (one GPU disposes of one video output).

(ii) *Transfer of results:* the visualization with OpenGL is impossible in the case of applying treatments on a set of images using one video output only. In this case, the transfer of results (output images) from GPU to CPU memory is required. This transfer time represents an additional cost for the application.

5. Hybrid Implementation on Heterogeneous Architectures

We presented in Section 3 the implementation details and steps of the proposed method of vertebra extraction on CPU. One disadvantage of this method is the computing time which increases significantly with the number of images and their resolution. Actually, the execution time of the edge detection is approximately 3 or 4 times greater than the time for histogram equalization and polygonal approximation. The ASM search procedure is not adapted for a parallel implementation due to the number of iterations which are dependent with each other. We proposed in [35] a solution based on the exploitation of the high power of GPUs in parallel. However, this solution does not exploit the CPU multiple computing units (cores) present in the majority of computers. Moreover, the solution based on GPU is hampered by the costs of data transfer between CPU and GPU memories. To reduce these constraints, we propose a parallel hybrid implementation which allows exploiting effectively the full computing power of heterogeneous architectures (multi-CPU/multi-GPU). This implementation is applied on the most intensive step of the vertebra segmentation method: edge detection. This section is presented in two parts: the first part describes our GPU implementation of edge detection step based on a recursive method. The second part describes the hybrid implementation of edge detection step on heterogeneous architectures.

5.1. GPU Implementation. This section describes the GPU implementation of edge detection step based on a recursive algorithm using Canny's design [2]. The noise truncature immunity and the reduced number of required operations make this method very efficient. This technique is based on four principale steps:

(i) recursive gradient computation (G_x, G_y).

(ii) gradient magnitude and direction computation.

(iii) non-maxima suppression.

(iv) hysteresis and thresholding.

We note that the recursive gradient computation step applies a Gaussian smoothing before filtering the image recursively using two Sobel filters in order to compute the gradients G_x and G_y. While the steps of gradient magnitude and direction computation, nonmaxima suppression, and hysteresis represent the same steps used for Canny filter described in Section 3.3.2.

```
1  for (i = 0; i < n; ++i) {\\ n: number of images
2   img = cvLoadImage (Input_image};
3   starpu_data_handle img_handle;
4   starpu_vector_data_register(&img_handle);
5   queue = add(queue, img, img_handle);
6  }
```

Listing 1: Loading of input images with StarPU.

The proposed GPU implementation of this recursive method is based on the parallelization of all the steps listed below on GPU using CUDA.

5.1.1. Recursive Gaussian Smoothing on GPU. The GPU implementation of the recursive Gaussian smoothing step is developed using the CUDA SDK individual sample package [41]. This parallel implementation is applied on Deriche recursive method [2]. The advantage of this method is that the execution time is independent of the filter width. The use of this technique for smoothing allows to have a better noise truncature immunity which represents an important requirement for our application.

5.1.2. Sobel Filtering on GPU. The recursive GPU implementation of this step is provided from the CUDA SDK individual sample package [41]. This parallel implementation exploits both shared and texture memories which allow to boost performance. This step applies a convolution of the source image by two Sobel filters of aperture size 3 in order to compute horizontal and vertical gradients G_x and G_y at each pixel. The GPU implementation is based firstly on a parallel horizontal convolution across the columns for computing G_x and secondly on a parallel vertical convolution across the lines for computing G_y.

5.1.3. Gradient Magnitude and Direction Computing on GPU. Once the horizontal and vertical gradients (G_x and G_y) have been computed, it is possible to calculate the gradient magnitude (intensity) using (4) and the gradient direction using (5). The CUDA implementation of this step is applied in parallel on image pixels, using a GPU grid computing containing a number of threads equal to image pixels number. Thus, each thread calculates the gradient magnitude and direction of one pixel of the image.

5.1.4. Nonmaxima Suppression on GPU. After computing the gradient magnitude and direction, we apply a CUDA function (kernel) which enumerates the local maxima (pixels with high gradient intensity) and deletes all nonridge pixels since local maxima are considered as a part of edges. We proposed to load the values of neighbors pixels (left, right, top, and bottom) in shared memory, since these values are required for the search of local maxima. The number of selected threads for parallelizing this step was also equal to image pixels number.

5.1.5. Hysteresis on GPU. Hysteresis represents the final step to product edges. It is based on the use of two thresholds T_1 and T_2. Any pixel in the image that has a gradient magnitude greater than T_1 is presumed to be an edge pixel and is marked as such immediately. Then, all the pixels connected to this edge pixel and that have a gradient intensity greater than T_2 are also selected as edge pixels. The GPU implementation of this step is achieved using the method described in [28]. Notice that we exploit also the GPU's shared memory for a fast loading of connected pixels values.

5.2. Hybrid Implementation. The GPU implementation described below allowed to improve considerably the performance of edge detection step in the case of processing one image only, since results can be visualized quickly with OpenGL [35]. However, if we apply treatments on a set of medical images (as required in our proposed method of vertebra detection), the transfer of results (output images) from GPU to CPU memory will be required. This transfer time represents an important cost for the application. Thus, we propose to implement the edge detection step on a set of medical images, by exploiting the full computing power of heterogeneous architectures (multi-CPU/multi-GPU) that enables to have faster solutions, with less transfer of data between CPU and GPU memories, as the images processed on CPU do not require any transfer. The proposed implementation is based on the executive support StarPU [34] which provides a unified runtime system for heterogeneous multicore architectures. Therefore, our hybrid implementation of the edge detection step applied on a set of X-ray images can be described in three steps: loading of input images, hybrid processing with StarPU, and updating and storing results.

5.2.1. Loading of Input Images. First, we have to load the input images in queues so that StarPU can apply treatments on images present on these queues. Listing 1 summarizes this step.

Line 2 allows loading the image in main memory, lines 3 and 4 enable to allocate a buffer (handle) StarPU which disposes of the loaded image address. Line 5 is used to add this image and the buffer StarPU in a queue that will contain all the images to treat.

5.2.2. Hybrid Processing with StarPU. Once the input images are loaded, StarPU can launch the CPU and GPU functions of edge detection (described, respectively, in Section 3.3.2

```
1   static starpu_codelet cl = {
2  .where = STARPU_CPU|STARPU_CUDA,   // CPU & GPU cores
3  .cpu_func = cpu_impl,     // define CPU fct
4  .cuda_func = cuda_impl,   // define GPU fct
5  .nbuffers = 1   // buffers number
6  };
```

Listing 2: The codelet StarPU.

```
1  while (queue != NULL)   {
2  task = starpu_task_create();    //Create task
3  task→cl = &cl;     //Define the codelet
4  task→handle = queue→img_handle;     //Define the buffer
5  task→handle.mode = STARPU_RW;     //Mode Read/Write
6  starpu_task_submit(task);     //Submit the task
7  queue = queue→next;     //Move to next image
8  }
```

Listing 3: Submission of tasks to the set of images.

and 5.1) on heterogeneous processing units (CPUs and GPUs). StarPU is based on two main structures: the codelet and the task. The codelet defines the computing units that could be exploited (CPUs or/and GPUs), and the related implementations (Listing 2). The StarPU tasks apply this codelet on the set of images.

In our case, each task is created and launched to treat one image in the queue. The scheduler of StarPU distributes automatically and effectively the tasks on the heterogeneous processing units. StarPU enables also an automatic transfer of data from CPU to GPU memory if tasks are executed on GPU. (Listing 3).

5.2.3. Updating and Storing Results. When all the StarPU tasks are completed, the results of GPU treatments must be repatriated in the buffers. This update is provided by a specific function in StarPU. This function enables also to transfer data from GPU to CPU memory in the case of treatments applied on GPU.

6. Experimental Results

6.1. Segmentation. The validation of the cervical mobility evaluation is made by the validation of the segmentation approach. If we can be sure to know exactly the contour of the vertebræ, we can efficiently evaluate the angles between them. In order to do this, we use a sample of 51 radiographs coming from the NHANES II database of the National Library of Medicine (http://archive.nlm.nih.gov/proj/dxpnet/nhanes/nhanes.php). These images are the digitized versions of X-ray films collected during 1976–1980. Persons aged 25 through 74 were examined. Interesting data in this work are radiographs focused on the cervical vertebræ. Actually, we study the 5 vertebral bodies C3 to C7. Note that the resolution is the same for all images, that is, 1763×1755 pixels. We then chose randomly 51 X-ray films allowing the visual presence of the vertebræ C3 to C7. This way, we can fix the test set to validate the segmentation method.

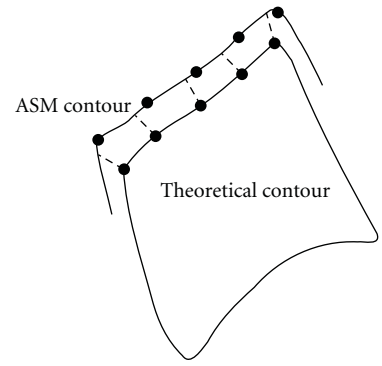


Figure 5: Point-to-line distance characterizing the error between a theoretical contour and an ASM-segmented contour.

One way to measure the segmentation error is to compute the distance between the ASM contour and a theoretical contour defined by a specialist. Therefore, a *gold standard* has been defined for the 51 radiographs of the test set. The chosen distance for measuring the segmentation error is the point-to-line distance. Used in [15, 16], the principle is to compute the length of the perpendicular dropped from each landmark of the theoretical contour to the spline evaluated between the landmarks of the ASM contour. A visual representation of the point-to-line distance is provided at the Figure 5.

Further in this paragraph, we present statistical results on the segmentation error. The reader will find the mean error (in px) for the sample of 51 radiographs, the median (in px), and finally the failure rate. These indicators are computed at each vertebra level (from C3 to C7). Let us remark that the segmentation error is given in pixels. However, the scanner used by the NLM to digitize the radiographs was of 146 dpi. Therefore, we can consider that 1 px is approximately equal to 0.2 mm. In order to determine the failure rate, we followed the example presented in [16]. The segmentation error is divided in success and failure distribution. Therefore, we consider as a failure any error greater than 3 standard deviations from the mean of the success distribution.

Before the analysis of the segmentation results, we need to measure the quality of the initialization based on the detection of the vertebræ in the radiograph. As we previously noticed, the goal of detecting corners in cervical spine radiographs is to initialize the mean shape of the ASM search. In [37], we showed the benefits of the polygonal approximation dedicated to the points of interest detection by comparing it with the Harris detector [42]. The Harris and Stephen's definition of a corner uses the information of the Hessian matrix of grey level intensities. This detector is based on the local autocorrelation matrix of a signal on a region defined around each point, which measures the local changes of the signal in different directions. However, the results show that in the particular context of cervical spine radiographs, the intensity gradient information is not useful for detecting the points of interest. Actually, we demonstrate the interest of using a geometrical definition of a corner for its detection. Another advantage of the polygonal approximation is that once the Canny parameters have been chosen, only one parameter remains to be fixed: the threshold ϵ. Furthermore, there is no influence between the Canny parameters and the one of the polygonal approximation.

In this section, we evaluate the influence of the initialization on the results. Table 1 shows the segmentation results with an initialization totally accomplished by a user. In fact, it was asked him to mark out manually all the vertebræ on the radiograph. We used a distinctive model for each vertebra level. In the literature [17–20], the models used are global ones. Their advantage is to bring information about the spine curvature, but they cannot efficiently accomplish a local segmentation. The only way to do this is to use a local vertebra model, but it requires a precise initialization close to the object of interest. The results of the Table 1 show the advantage to use such a model. The segmentation error is about 2.90 px and the percentage of failures is more than acceptable for each vertebra level (compared to the literature, see, for instance [15, 17–20]).

We could have stopped the experiments here, but it is not conceivable to ask a user to mark out all the vertebræ on every image he has to segment. For this reason, one of the contributions of this paper is to propose a semi-automatization of the ASM initialization. The data related to the Table 2 present the results based on this automated initialization. The analysis of the table demonstrates two particular trends. First, if we consider the mean segmentation error, we notice that its value is slightly increased in comparison with the Table 1. A meticulous analysis permits to target the step of the procedure responsible for this phenomenon. In fact, the results degradation is due to the points of interest detection by the polygonal approximation. Nevertheless, this effect is minimal if we look at the results of the Table 2.

Table 1: Statistical results on the error segmentation: local vertebra model (manual initialization).

Vert.	Mean (px)	Median (px)	Fail. (%)
C3	2.95	2.30	7.84
C4	2.63	2.43	1.96
C5	2.74	2.20	3.92
C6	2.98	2.65	3.92
C7	3.11	2.54	1.96

Table 2: Statistical results on the error segmentation: local vertebra model (automated initialization).

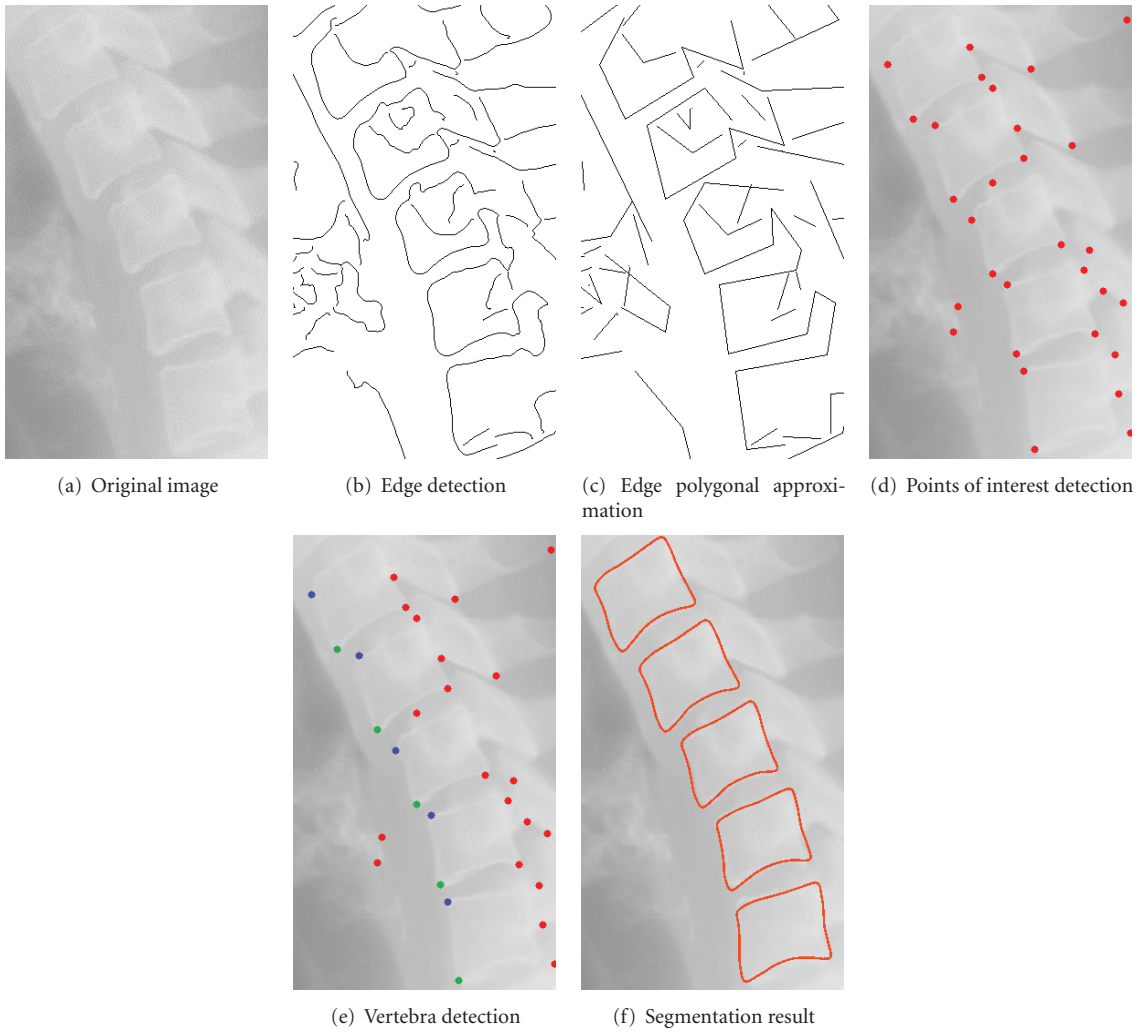
Vert.	Mean (px)	Median (px)	Fail. (%)
C3	2.97	2.36	7.84
C4	3.74	2.42	7.84
C5	2.86	2.34	5.88
C6	3.48	2.73	9.80
C7	3.27	2.50	5.88

A particular limitation of our approach could arise in a specific case. If two vertebræ are merged, the corner detection could confuse a higher corner of a vertebra with the lower corner and the adjacent vertebra. Finally, we give the user a visual illustration of the whole framework to perform the vertebra segmentation at the Figure 6.

6.2. Performance. On the one hand, we can say that the quality of the vertebra segmentation remains identical since the procedure has not changed. Only the architecture and the implementation did. On the other hand, the exploitation of heterogeneous architectures (multi-CPU/multi-GPU) in parallel for vertebra extraction enabled to accelerate the computation time. This acceleration is due to the hybrid implementation for edge detection step based on a recursive method using Deriche-Canny method. This fact allowed to apply our proposed method on large sets of X-ray medical images in order to have more precision for vertebra detection results.

Figure 7(a) presents the comparison of the computing times between sequential (CPU), parallel (GPU), and hybrid (multi-CPU/multi-GPU) implementations of the edge detection step, applied on a set of 200 images using different resolutions. Figure 7(b) shows the speedups obtained with these implementations. The accelerations presented at Figure 7 are due to two level of parallelism: a low-level and a high-level parallelization.

(i) A low-level parallelization by porting the edge detection step on GPU (parallel processing between pixels in image: intraimage parallel processing).



(a) Original image (b) Edge detection (c) Edge polygonal approximation (d) Points of interest detection

(e) Vertebra detection (f) Segmentation result

Figure 6: Illustration of the whole framework for the segmentation.

(ii) A high-level parallelization (interimages parallel processing) enabling to exploit simultaneously both CPUs and GPUs cores so that each core treats a subset of images.

Experimentations have been conducted on several platforms, that is, GPU Tesla C1060 and CPU Dual core:

(i) CPU: Dual Core 6600, 2.40 GHz, 2 GB RAM of Memory.

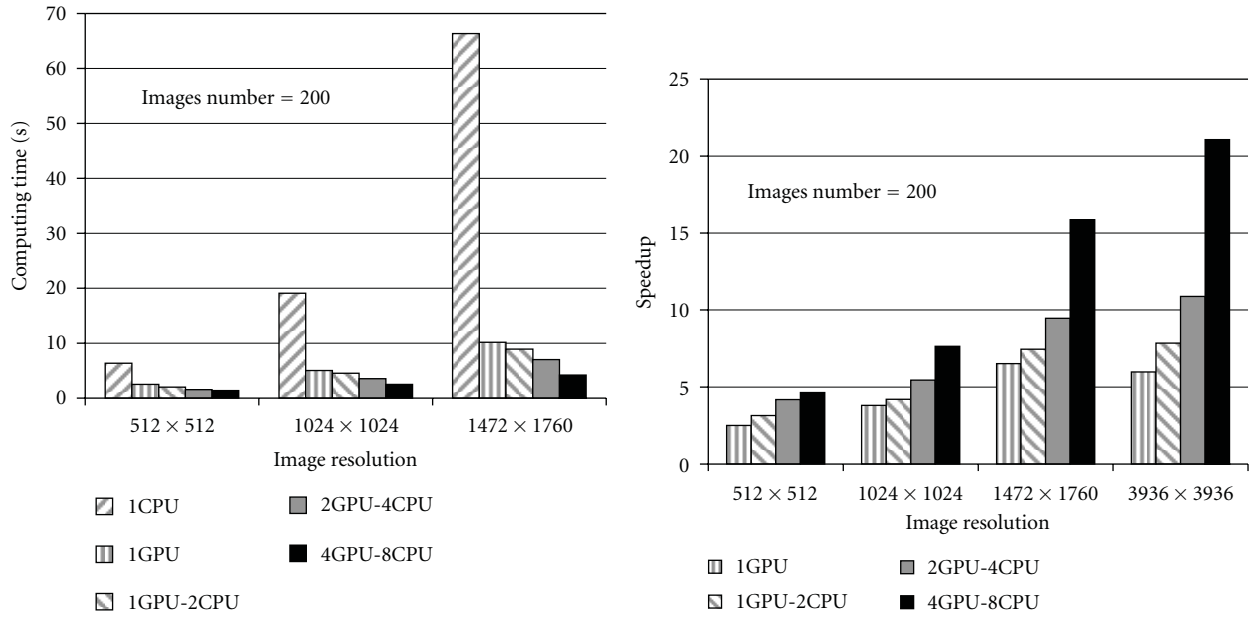
(ii) GPU: NVIDIA Tesla C1060, 240 CUDA cores, 4 GB of Memory.

7. Conclusion

In this paper, we proposed a framework for vertebra segmentation based on a local statistical model. An original process in order to locate vertebræ in a radiograph has been developed. The principle is to detect features characterizing the vertebræ: the anterior corners. The extraction procedure is composed of 4 steps: a contrast-limited adaptive histogram equalization to improve the image contrast, an edge detection, an anterior corner detection, and finally the vertebra localization.

Generally, the computation time and noise immunity truncature represent the most important requirements in medical image processing and specifically for our application. The graphics processors provided a solution by exploiting the GPU's computing units in parallel. However, this solution is hampered by the costs of data transfers between CPU and GPU memories. Thus, we proposed a parallel hybrid implementation of the recursive edge method using Deriche-Canny approach. This implementation allowed to exploit the full computing power of heterogeneous architectures. Moreover, this solution requires a less transfer of data between CPU and GPU memories, as the treatments on CPU do not require any transfer.

As future work, we plan to develop a fully automatic segmentation approach based on a learning method such as support vector machine (SVM). The main issue is to find an efficient descriptor to train the supervised model. We also aim to provide an automatic parallel implementation



(a) Computing time of edge detection step using hybrid platforms

(b) Speedup obtained for edge detection step using hybrid platforms

Figure 7: Performance of recurive edge detection using heterogeneous architectures.

exploiting the full computing power of hybrid architectures. This implementation could choose automatically the processing units for each step of our medical application. Thus, the most intensive steps (initialization: edge detection) would be implemented on heterogeneous platforms (multi-CPU/multi-GPU), and the less intensive or not parallelizable steps (learning, modelization, and segmentation) would exploit the CPU multiple cores (multi-CPU).

Acknowledgments

The authors would like to thank the Communauté Française de Belgique for supporting this work under the ARC-OLIMP Research Project, Grant no. AUWB-2008-13-FPMs11. They aknowledge also the support of the European COST action IC0805 "Open European Network for High Performance Computing on Complex Environment".

References

[1] F. Puglisi, R. Ridi, F. Cecchi, A. Bonelli, and R. Ferrari, "Segmental vertebral motion in the assessment of neck range of motion in whiplash patients," *International Journal of Legal Medicine*, vol. 118, no. 4, pp. 235–239, 2004.

[2] R. Deriche, "Using Canny's criteria to derive a recursively implemented optimal edge detector," *International Journal of Computer Vision*, vol. 1, no. 2, pp. 167–187, 1987.

[3] X. Siebert, S. Dupont, P. Fortemps, and D. Tardieu, "Mediacycle: browsing and performing with sound and image libraries," in *QPSR of the Numediart Research Program*, T. Dutoit and B. Macq, Eds., vol. 2, pp. 19–22, 2009.

[4] C. Chevrefils, F. Chériet, C.-E. Aubin, and G. Grimard, "Texture analysis for automatic segmentation of intervertebral disks of scoliotic spines from MR images," *IEEE Transactions on Information Technology in Biomedicine*, vol. 13, no. 4, pp. 608–620, 2009.

[5] S.-H. Huang, Y.-H. Chu, S.-H. Lai, and C. L. Novak, "Learning-bBased vertebra detection and iterative normalized-cut segmentation for spinal MRI," *IEEE Transactions on Medical Imaging*, vol. 28, no. 10, Article ID 4967966, pp. 1595–1605, 2009.

[6] M. Aslan, A. Ali, H. Rara et al., "A novel 3d segmentation of vertebral bones from volumetric ct images using graph cuts," in *Advances in Visual Computing*, vol. 5876 of *Lecture Notes in Computer Science*, pp. 519–528, Springer, Berlin, Germany, 2009.

[7] T. Klinder, J. Ostermann, M. Ehm, A. Franz, R. Kneser, and C. Lorenz, "Automated model-based vertebra detection, identification, and segmentation in CT images," *Medical Image Analysis*, vol. 13, no. 3, pp. 471–482, 2009.

[8] J. Weese, M. Kaus, C. Lorenz, S. Lobregt, R. Truyen, and V. Pekar, "Shape constrained deformable models for 3D medical image segmentation," in *Information Processing in Medical Imaging*, vol. 2082 of *Lecture Notes in Computer Science*, Springer, Berlin, Germany, 2001.

[9] H. Shen, A. Litvin, and C. Alvino, "Localized priors for the precise segmentation of individual vertebras from ct volume data," in *Medical Image Computing and Computer-Assisted Intervention MICCAI 2008*, vol. 5241 of *Lecture Notes in Computer Science*, pp. 367–375, Springer, Berlin, Germany, 2008.

[10] S. Tan, J. Yao, M. M. Ward, L. Yao, and R. M. Summers, "Level set based vertebra segmentation for the evaluation of ankylosing spondylitis," in *Medical Imaging: Image Processing*, vol. 6144 of *Proceedings of SPIE*, pp. 58–67, San Diego, Calif, USA, February 2006.

[11] M. Benjelloun and S. Mahmoudi, "X-ray image segmentation for vertebral mobility analysis," *International Journal of Computer Assisted Radiology and Surgery*, vol. 2, no. 6, pp. 371–383, 2008.

[12] T. F. Cootes and C. J. Taylor, "Active shape models—'smart snakes'," in *Proceedings of the British Machine Vision Conference*, pp. 266–275, Springer, Berlin, Germany, 1992.

[13] T. F. Cootes, G. J. Edwards, and C. J. Taylor, "Active appearance models," in *5th European Conference on Computer Vision*, pp. 484–498, Springer, Berlin, Germany, 1998.

[14] L. R. Long and G. R. Thoma, "Use of shape models to search digitized spine X-rays," in *13th IEEE Symposium on Computer-Based Medical Systems*, pp. 46–50, IEEE Computer Society, 2000.

[15] P. P. Smyth, C. J. Taylor, and J. E. Adams, "Automatic measurement of vertebral shape using active shape models," *Image and Vision Computing*, vol. 15, no. 8, pp. 575–581, 1997.

[16] P. P. Smyth, C. J. Taylor, and J. E. Adams, "Vertebral shape: Automatic measurement with active shape models," *Radiology*, vol. 211, no. 2, pp. 571–578, 1999.

[17] M. G. Roberts, T. F. Cootes, and J. E. Adams, "Linking sequences of active appearance sub-models via constraints: an application in automated vertebral morphometry," in *Proceedings of the 14th British Machine Vision Conference*, pp. 349–358, Norwich, UK, 2003.

[18] M. G. Roberts, T. F. Cootes, and J. E. Adams, "Vertebral shape: automatic measurement with dynamically sequenced active appearance models," in *Medical Image Computing and Computer-Assisted Intervention-MICCAI 2005*, vol. 3750 of *Lecture Notes in Computer Science*, pp. 733–744, Springer, Berlin, Germany, 2005.

[19] M. G. Roberts, T. F. Cootes, and J. E. Adams, "Automatic segmentation of lumbar vertebrae on digitised radiographs using linked active appearance models," in *Proceedings of the Medical Image Understanding and Analysis Conference*, pp. 120–124, Manchester, UK, July 2006.

[20] M. G. Roberts, T. F. Cootes, E. Pacheco, T. Oh, and J. E. Adams, "Segmentation of lumbar vertebrae using part-based graphs and active appearance models," in *Proceedings of the 12th International Conference on Medical Image Computing and Computer-Assisted Intervention*, pp. 1017–1024, Springer, 2009.

[21] G. Zamora, H. Sari-Sarraf, and L. R. Long, "Hierarchical segmentation of vertebrae from x-ray images," in *Proceedings of the Medical Imaging 2003: Image Processing*, vol. 5032 of *Proceedings of SPIE*, pp. 631–642, San Diego, Calif, USA, February 2003.

[22] X. Dong and G. Zheng, "Automated vertebra identification from X-ray images," in *Image Analysis and Recognition*, vol. 6112 of *Lecture Notes in Computer Science*, pp. 1–9, 2010.

[23] Z. Yang, Y. Zhu, and Y. pu, "Parallel image processing based on CUDA," in *Proceedings of the International Conference on Computer Science and Software Engineering*, pp. 198–201, Wuhan, China, December 2008.

[24] J. Fung, S. Mann, and C. Aimone, "OpenVIDIA : parallel gpu computer vision," in *Proceedings of the ACM Multimedia*, pp. 849–852, Hilton, Singapore, November 2005.

[25] OpenGL, "OpenGL Architecture Review Board: ARB vertex program. Revision 45," 2004, http://oss.sgi.com/projects/ogl-sample/registry.

[26] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Transactions on Graphics*, vol. 22, pp. 896–907, 2003.

[27] NVIDIA, "NVIDIA CUDA," 2007, http://www.nvidia.com/cuda.

[28] Y. M. Luo and R. Duraiswami, "Canny edge detection on NVIDIA CUDA," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, Anchorage, Alaska, USA, June 2008.

[29] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–698, 1986.

[30] Y. Heng and L. Gu, "GPU-based volume rendering for medical image visualization," in *Proceedings of the 27th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pp. 5145–5148, Shanghai, China, September 2005.

[31] M. Smelyanskiy, D. Holmes, J. Chhugani et al., "Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1563–1570, 2009.

[32] T. Schiwietz, T. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," in *Proceedings of the Medical Imaging: Visualization, Image-Guided Procedures, and Display*, Proceedings of SPIE, pp. 646–655, San Diego, Calif, USA, March 2006.

[33] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Orti, "An extension of the starSs programming model for platforms with multiple GPUs," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing (Euro-Par '09)*, pp. 851–862, 2009.

[34] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," in *Concurrency and Computation: Practice and Experience, Euro-Par 2009*, pp. 863–874, 2009.

[35] S. A. Mahmoudi, F. Lecron, P. Manneback, M. Benjelloun, and S. Mahmoudi, "GPU-based segmentation of cervical vertebra in X-ray images," in *Proceedings of the High-Performance Computing on Complex Environments Workshop, in conjunction with the IEEE International Conference on Cluster Computing*, pp. 1–8, 2010.

[36] C. Goodall, "Procrustes methods in the statistical analysis of shape," *Journal of the Royal Statistical Society. Series B*, vol. 53, no. 2, pp. 285–339, 1991.

[37] F. Lecron, M. Benjelloun, and S. Mahmoudi, "Points of interest detection in cervical spine radiographs by polygonal approximation," in *Proceedings of the 2nd International Conference on Image Processing Theory, Tools and Applications*, pp. 81–86, IEEE Computer Society, 2010.

[38] S. M. Pizer, E. P. Amburn, J. D. Austin et al., "Adaptive histogram equalization and its variations," *Computer Vision, Graphics, and Image Processing*, vol. 39, no. 3, pp. 355–368, 1987.

[39] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica*, vol. 10, no. 2, pp. 112–122, 1973.

[40] GPU4VISION, "GPU4VISION," 2010, http://www.gpu4vision.org.

[41] NVIDIA, "NVIDIA CUDA SDK code samples," http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

[42] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proceedings of the 4th Alvey vision conference*, vol. 15, pp. 147–151, Manchester, UK, 1988.