FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
INSTITUT FÜR INFORMATIK (MATHEMATISCHE MASCHINEN UND DATENVERARBEITUNG)

Lehrstuhl für Informatik 10 (Systemsimulation)

A parallel K-SVD implementation for CT image denoising

D. Bartuschat, A. Borsdorf, H. Köstler, R. Rubinstein, M. Stürmer

Lehrstuhlbericht 09-01

A parallel K-SVD implementation for CT image denoising

D. Bartuschat, A. Borsdorf, H. Köstler, R. Rubinstein, M. Stürmer

Abstract

In this work we present a new patch-based approach for the reduction of quantum noise in CT images. It utilizes two data sets gathered with information from the odd and even projections respectively that exhibit uncorrelated noise for estimating the local noise variance and performs edge-preserving noise reduction by means of the K-SVD algorithm. It is an efficient way for designing overcomplete dictionaries and finding sparse representations of signals from these dictionaries. For image denoising, the K-SVD algorithm is used for training an overcomplete dictionary that describes the image content effectively. K-SVD has been adapted to the non-gaussian noise in CT images. In order to achieve close to real-time performance we parallelized parts of the K-SVD algorithm and implemented them on the Cell Broadband Engine Architecture (CBEA), a heterogenous, multicore, distributed memory processor. We show denoising results on synthetic and real medical data sets.

1 Introduction

A major field in CT research has been the investigation of approaches for image noise reduction. Reducing noise corresponds to an increase of the signal-to-noise ratio (SNR). Consequently, the patient's x-ray exposure can be reduced, since a smaller radiation dose sufficies for acquiring the x-ray projection data, from which CT images are then reconstructed.

The main source of noise in CT is quantum noise. It results from statistical fluctuations of x-ray quanta reaching the dectector. Noise in projection data is known to be poisson-distributed. However, during the reconstruction process, by means of the (most common) filtered backprojection method, noise distribution is changed. Due to complicated dependencies of noise on scan parameters and on spatial position [15], noise distribution in the final CT image is usually unknown and noise variance in CT images is spatially changing. Additionally, strong directed noise in terms of streak artifacts may be present [13].

Several different algorithmic approaches for CT noise reduction exist, like methods that suppress noise in projection data before image reconstruction. Other algorithms reduce noise during CT reconstruction by optimizing statistical objective functions [17, 16]. Another common area of research includes the development of algorithms for noise reduction in reconstructed CT images. These methods are required to reduce noise while preserving edges, as well as small structures, that might be important for diagnosis. Standard edge-preserving methods in the spatial domain are partial differential equation (PDE) based methods [20, 22]. In the frequency domain, wavelet-based methods for denoising are prevalently investigated [9, 6].

We proposed CT denoising methods that deal with the complicated noise properties by utilizing two spatially identical images containing uncorrelated noise. In order to differentiate between structure and noise in a certain region, correlation can be computed and used as a measure for similarity, what has been done for wavelet based methods in [3, 5]. Additionally, position dependent noise estimation can be performed by computing the noise variance, as in [4]. These wavelet based methods are compared to PDE based denoising methods with nonlinear isotropic and anisotropic diffusion[18, 19].

In this work, we present a new patch-based approach for the reduction of quantum noise in CT images. It utilizes again the idea of using two data sets with uncorrelated noise for estimating the local noise variance and performs edge-preserving noise reduction by means of the K-SVD algorithm that is an efficient algorithm for designing overcomplete dictionaries and then finding sparse representations of signals from these dictionaries [11]. For image denoising, the K-SVD algorithm is used for training an overcomplete dictionary that describes the image content effectively. This method performs very well for removing additive gaussian noise from images and has been adapted to the non-gaussian noise in CT images.

In order to achieve close to real-time performance, the algorithm is – in addition to a C++ implementation – also parallelized on the Cell Broadband Engine Architecture (CBEA), a heterogenous, multicore, distributed memory processor. Its special architecture explicitly parallelizes computations and transfer of data and instructions. For efficient code, this kind of parallelism has to be exploited, together with data-level parallelism in the form of vector processing and thread-level parallelism by means of software threads.

We introduce the idea of sparse representations and the K-SVD algorithm in section 2 and show how it can be used for CT image denoising. In section 3 we discuss the Cell implementation and in section 4 we present qualitative and quantitative denoising and performance results both for synthetic and real medical data sets. Our work is summarized in section 5.

2 Methods

In the following, we describe sparse representations of signals, and methods for finding these, given an overcomplete dictionary, which contains a set of elementary signals.

In combination with a dictionary representing the noise-free part of a noisy signal, sparse representations can be used for signal denoising. A dictionary that has this property, can be obtained by performing dictionary training with the K-SVD algorithm. The advantage of this algorithm is, that it can be used for adapting the dictionary to the image to be denoised. This trained dictionary can then be used for denoising the image, on which it was trained. This is possible, since stationary Gaussian white noise is not learned by that dictionary.

2.1 Sparse Representations and the OMP Algorithm

By extending a set of elementary vectors beyond a basis of the signal vector space, signals can be compactly represented by a linear combination of only few of these vectors. The problem of finding the sparsest representation of a signal $\mathbf{y} \in \mathbb{R}^n$ among infinitely many solutions of this underdetermined problem reads as

$$\min_{\mathbf{a}} \|\mathbf{a}\|_0 \text{ subject to } \mathbf{D}\mathbf{a} = \mathbf{y}, \tag{1}$$

where $\|\cdot\|_0$ denotes the ℓ_0 - seminorm that counts the nonzero entires of a vector $\|\mathbf{a}\|_0 = \sum_{j=0}^{K} |\mathbf{a}_j|^0$. The given full rank matrix $\mathbf{D} \in \mathbb{R}^{n \times K}$ represents the overcomplete dictionary, which has a higher number of atoms K than the dimension of the atoms n. For known error tolerance ϵ, eq.(1) can be reformulated as

$$\min_{\mathbf{a}} \|\mathbf{a}\|_0 \text{ subject to } \|\mathbf{D}\mathbf{a} - \mathbf{y}\|_2 \leq \epsilon. \tag{2}$$

This sparsest approximation allows a more compact representation of signals. It can be deployed for removing noise from signals, if ϵ fits the present noise.

OMP Algorithm: Exactly determining the sparsest representation of signals is an NP-hard combinatorial problem [8]. A simple greedy algorithm, the orthogonal matching pursuit (OMP) algorithm [8] depicted in algorithm 1, is used for performing sparse-coding approximately by sequentially selecting dictionary atoms.

Algorithm 1 OMP algorithm: $\mathbf{a} = \text{OMP}(\mathbf{D}, \mathbf{x}, \epsilon \textit{ or } L)$

1 Init: Set $\mathbf{r}_0 = \mathbf{x}$, $j = 0$, $I = \emptyset$
2 **while** $j < L$ && $\|\mathbf{r}\|_2^2 > \epsilon$ **do**
3 $\quad \mathbf{p} = \mathbf{D}^T \mathbf{r}_{j-1}$
4 $\quad \hat{k} = \underset{k}{\text{argmax}} |\mathbf{p}|$
5 $\quad I = I \cup \hat{k}$
6 $\quad \mathbf{a}_I = \mathbf{D}_I^+ \mathbf{x}$
7 $\quad \mathbf{r}_j = \mathbf{x} - \mathbf{D}_I \mathbf{a}_I$
8 $\quad j = j + 1$
9 **end while**

Here, I is a data structure for storing a sequence of chosen atoms' indices. $\mathbf{D}_I$ denotes a matrix that comprises only those atoms of the dictionary, which have been selected in the previous steps. In line 3, the projection of the residual on the dictionary is computed. After that, the greedy selection step is performed in line 4. It selects the index $\hat{k}$ of the largest element of $\mathbf{p}$, which corresponds to the the atom with maximal correlation to the residual. Having added the new atom index to the set I, the orthogonalization step in line 6 follows. It ensures that all selected atoms are linearly independent [8]. Here, $\mathbf{D}_I^+$ denotes the pseudo-inverse of $\mathbf{D}_I$. In step 7, the new residual is computed, which is orthogonal to all previously selected atoms. When the stopping criterion is fulfilled, the coefficients have been found and the algorithm terminates. The stopping criterion is fulfilled, if either the sparse representation error is smaller than the error tolerance, or the maximum number of atoms L has been found.

OMP is guaranteed to converge in finite-dimensional spaces in a finite number of iterations [8]. It consecutively removes those signal components, that are highly correlated to few dictionary atoms. At the first iterations, the representation error decays quickly, as long as highly correlated atoms have not yet been chosen. Later, when all similar atoms have been selected, the residual decays only slowly and the residuals behave like realizations of white noise [8]. Therefore, signal approximation is truncated, depending on the error tolerance ϵ of (2).

Batch-OMP Algorithm: The Batch-OMP algorithm [21], summarized in algorithm 2, accelerates the OMP algorithm for large sets of signals by two techniques: It replaces the computation of the pseudoinverse in the orthogonalization step, which is done in OMP by a singular value decomposition (SVD), by a progressive Cholesky update. And it performs pre-computation of the gram matrix $\mathbf{G} = \mathbf{D^T D}$, which results in lower-cost computations at each iteration.

The orthogonalization step and the following residual update in the OMP algorithm of the previous section, can be written as

$$\mathbf{r} = \mathbf{x} - \mathbf{D}_I (\mathbf{D}_I^T \mathbf{D}_I)^{-1} \mathbf{D}_I^T \mathbf{x}. \tag{3}$$

Due to the orthogonalization step, the matrix $(\mathbf{D}_I^T \mathbf{D}_I)$ stays non-singular. This matrix is symmetric positive definite (SPD) and can therefore be decomposed by means of a Cholesky decomposition. To this matrix, a new row and column are added at each iteration, since a

Algorithm 2 $\mathbf{a}$ = Batch-OMP $\left(\mathbf{p}^0 = \mathbf{D}^T\mathbf{x},\ \epsilon^0 = \mathbf{x}^T\mathbf{x},\ \mathbf{G} = \mathbf{D}^T\mathbf{D}\right)$

1 Init: Set $I = \emptyset$, $\mathbf{L} = [1]$, $\mathbf{a} = 0$, $\delta^0 = 0$, $\mathbf{p} = \mathbf{p}^0$, $n = 1$
2 **while** $\epsilon^{n-1} > \epsilon$ **do**
3 $\quad\hat{k} = \underset{k}{\operatorname{argmax}} |\mathbf{p}_{n-1}|$
4 $\quad I^n = I^{n-1} \cup \hat{k}$
5 $\quad$ **if** $n > 1$ **then**
6 $\quad\quad\mathbf{w} = \text{Solve for } \mathbf{w} \left\{\mathbf{L}^n\mathbf{w} = \mathbf{G}_{I^n,\hat{k}}\right\}$
7 $\quad\quad\text{where } \mathbf{L}^n = \begin{bmatrix} \mathbf{L}^{n-1} & 0 \\ \mathbf{w}^T & \sqrt{1-\mathbf{w}^T\mathbf{w}} \end{bmatrix}$
8 $\quad$ **end if**
9 $\quad\mathbf{a}_{I^n} = \text{Solve for } \left\{\mathbf{L}^n\left(\mathbf{L}^n\right)^T\mathbf{a}_{I^n} = \mathbf{p}^0_{I^n}\right\}$
10 $\quad\beta = \mathbf{G}_{I^n}\mathbf{a}_{I^n}$
11 $\quad\mathbf{p}_n = \mathbf{p}^0 - \beta$
12 $\quad\delta^n = \mathbf{a}^T_{I^n}\beta_{I^n}$
13 $\quad\epsilon^n = \epsilon^{n-1} - \delta^n + \delta^{n-1}$
14 $\quad n = n + 1$
15 **end while**

new atom is there added to $\mathbf{D}_I$. Thus, a new row is added at each iteration to its decomposed lower triangular Cholesky matrix $\mathbf{L}$.

The residual in the OMP algorithm does not need to be computed explicitly at each iteration. Instead, only $\mathbf{D}^T\mathbf{r}$, the projection of the residual on the dictionary, is required. This can be exploited by directly computing $\mathbf{D}^T\mathbf{r}$ instead of the residual, yielding

$$\mathbf{p} = \mathbf{D}^T\mathbf{r} = \mathbf{p}^0 - \mathbf{G}_I\left(\mathbf{D}_I\right)^+\mathbf{x} \tag{4}$$
$$= \mathbf{p}^0 - \mathbf{G}_I(\mathbf{G}_{I,I})^{-1}\mathbf{p}^0_I. \tag{5}$$

with $\mathbf{p}^0 = \mathbf{D}^T\mathbf{x}$. This is done in line 11 of algorithm 2.
One index I at a matrix indicates that only a sub-matrix is considered containing the columns of the matrix which are indexed by I. For a vector this means that only those elements are selected, which are indexed by I. Two indices I, I denote that the sub-matrix consists of indexed rows and columns.
From this, the projection can be computed without explicitly computing $\mathbf{r}$. The update step requires now only multiplication by the $\mathbf{G}_I$, which can be selected from the pre-computed matrix $\mathbf{G}$. The matrix $\mathbf{G}_{I,I}$ is inverted using the progressive Cholesky update yielding $\mathbf{L}^n\left(\mathbf{L}^n\right)^T$. This progressive Cholesky update is performed in lines 5 – 8. The new row of the decomposed lower triangular Cholesky matrix is computed from the previous lower triangular Cholesky matrix, and a vector that contains gram matrix entries from the $\hat{k}$-th column of the gram matrix and rows corresponding to the previously selected atoms I, by means of forward substitution. For more details, see [21].
The nonzero element coefficient vector $\mathbf{a}_{I^n}$ can then be computed from the formula

$$\mathbf{L}^n\left(\mathbf{L}^n\right)^T\mathbf{a}_{I^n} = \mathbf{p}^0_{I^n}, \tag{6}$$

by means of a forward- and backward substitution, as done in line 9. Here, n denotes the iteration counter.

However, when the OMP is used for solving an error-constrained sparse approximation problem, the residual is required to check the termination criterion. Therefore, an incremental formula for the ℓ^2 norm of the residual has been derived in [21]. This formula is used in line

13 and the previous line to compute the residual norm. The stopping criterion is checked in line 2.

It has been shown in [10], that for the presence of small amounts of noise, greedy algorithms for computing sparse representations are locally stable. This holds, if the dictionary is mutually incoherent and when it offers sufficiently sparse representations for the ideal noiseless signal. The *mutual coherence* $m(\mathbf{D})$ of a dictionary $\mathbf{D}$ is a measure for the similarity or linear dependency of its atoms. It is defined as the maximal absolute scalar product between two different normalized atoms of the dictionary:

$$m(\mathbf{D}) = \max_{i \neq j} \frac{d_i^T d_j}{d_i^T d_i d_j^T d_j}. \tag{7}$$

Dictionaries are called *incoherent*, if $m(\mathbf{D})$ is small. Under these conditions, these algorithms recover the ideal sparse signal representation, with an error that grows at most proportionally to the noise level.

2.2 K-SVD Algorithm

K-SVD can be seen as a generalization of the K-means algorithm, that is commonly used for vector quantization [2]. Given a set of N training signals in a matrix $\mathbf{Y} \in \mathbb{R}^{n \times N}$, the K-SVD method searches for the dictionary $\mathbf{D}$ that best represents these signals.
It factors the matrix $\mathbf{Y}$ into the dictionary $\mathbf{D} \in \mathbb{R}^{n \times K}$, which contains the K dictionary atoms, and into the sparse coefficient matrix $\mathbf{A} \in \mathbb{R}^{K \times N}$, which comprises sparse representation coefficients for each of the N signals $\mathbf{y}$

$$\min_{D,A} \sum_i \|\mathbf{a}_i\|_0 \text{ subject to } \|\mathbf{DA} - \mathbf{Y}\|_F^2 \leq \epsilon. \tag{8}$$

Here, $\mathbf{a}_i$ denotes the sparse representation vector corresponding to the i-th training signal, ϵ denotes the error tolerance.

The K-SVD algorithm [2] consists of two steps. In the first step – the *Sparse Coding Stage* – it finds sparse representation vectors $\mathbf{a}_i$. Any pursuit algorithm can be used to compute $\mathbf{a}_i$ for all training signals. The second step of the K-SVD algorithm – the *Dictionary Update Stage* – updates the dictionary such that it best represents the training signals for the found coefficients in $\mathbf{A}$. Each of the K columns k in $\mathbf{D}$ is updated by the following steps of the dictionary update:

- Find the set of training signals that use the current atom, defined by $\omega_k = \{i \mid 1 \leq i \leq K, a_T^k(i) \neq 0\}$. Here, $\mathbf{a}_T^k$ denotes the kth row in $\mathbf{A}$, which corresponds to the atom that is updated.

- Compute the overall sparse representation error matrix $\mathbf{E}_k$. Each of the N columns of this matrix stands for the error of the corresponding training signal, when the kth atom is removed. It is computed by $\mathbf{E}_k = \mathbf{Y} - \sum_{j \neq k} \mathbf{d}_j \mathbf{a}_T^j$.

- In order to ensure sparsity when computing the representation vectors in the next step, restrict $\mathbf{E}_k$ by choosing only those columns that correspond to signals, which use the current atom. I.e. choose only columns whose indices are contained in ω_k. The restricted matrix is denoted by $\mathbf{E}_k^R$.

- Apply an SVD decomposition to obtain $\mathbf{E}_k^R = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. The first column of $\mathbf{U}$ is assigned to the dictionary atom $\mathbf{d}_k$. The coefficient vector is updated to be the first column of V multiplied by the largest singular value $\mathbf{\Sigma}(1,1)$.

The SVD finds the closest rank-1 matrix that approximates $\mathbf{E}_k$. By assigning the elements of this approximation to $\mathbf{d}_k$ and $\mathbf{a}_T^k$ as shown before, the sparse representation error is minimized.

By performing several iterations of these two steps, where the first step computes the coefficients of the given dictionary, and the second step updates the dictionary, the dictionary is adapted to the training signals. The convergence of the K-SVD algorithm is guaranteed, if the sparse coding stage is performed perfectly. In this case, the total representation error $\|\mathbf{DA} - \mathbf{Y}\|_F^2$ is decreased in each sparse coding step. In addition, the mean squared error (MSE) can be reduced in the dictionary update stage, otherwise it does not change there. The sparsity constraint is not violated. Thus, convergence of the K-SVD algorithm to a local minimum is guaranteed, if the pursuit algorithm robustly finds a solution.

The approximate K-SVD algorithm (see Algorithm 3, [21]) starts with an initial dictionary, which is improved in the following k iterations 4. In line 5, sparse representation coefficients for each signal are computed by means of the OMP algorithm or any other suitable algorithm for the current dictionary. Then, each of the K dictionary atoms is updated by a numerically cheaper approximation than the one described above using SVD computation.
The first improvement in terms of numerical costs is, that it computes the error matrix only for those signals, which use the atom to be updated. As can be seen in line 7, the error matrix needs not to be computed explicitly. The set of indices of those signals, which corresponds to ω_k from the previous section, is for simplicity denoted by I.
The update of the dictionary atoms is performed by optimizing $\|\mathbf{E}_I\|_F^2 = \|\mathbf{DA}_I - \mathbf{Y}_I\|_F^2$, over the dictionary atom and the coefficient vector

$$\{\mathbf{d_k}, \mathbf{a}_T^k\} = \underset{\mathbf{d}_k, \alpha_k}{\operatorname{argmin}} \|\mathbf{E}_{I,k} - \mathbf{d}_k \alpha_k^T\|_F^2 \text{ subject to } \|\mathbf{d}_k\|_2 = 1. \tag{9}$$

For simplicity, $\left(\mathbf{a}_{T,I}^k\right)^T$ is denoted by α_k.
The second improvement in terms of numerical costs is that this problem is solved in the approximate K-SVD algorithm by means of one iteration of a numeric power method. The dictionary atom is obtained by

$$\mathbf{d}_k = \frac{\mathbf{E}_{I,k}\alpha_k}{\|\mathbf{E}_{I,k}\alpha_k\|_2}. \tag{10}$$

This is done in line 7. After having normalized the dictionary atom with respect to the ℓ^2 norm in line 8., the nonzero coefficients are computed by

$$\alpha_k = \mathbf{E}_{I,k}^T \mathbf{d}_k \tag{11}$$

in line 9. These coefficients are needed for the update of the next dictionary atom. As mentioned before, the dictionary update step converges not to a global solution, but a local solution. Thus, the approximate solution is sufficient, even if the power method is truncated after one iteration.

2.3 Extension of K-SVD Algorithm to CT Image Denoising

The sparse approximation problem and the K-SVD algorithm are applied to remove noise from a given image y and recover the original noise-free image x that is corrupted with additive,

Algorithm 3 Approximate K-SVD algorithm

Input: Training Signals $\mathbf{Y}$, initial dictionary $\mathbf{D}_0$, sparse representation error tolerance ϵ , number of iterations k
Output: Dictionary $\mathbf{D}$ and sparse coefficient matrix $\mathbf{A}$
Init: Set $\mathbf{D} = \mathbf{D}_0$
for $n = 1...k$ **do**
  $\forall i : \mathbf{A}_i = \underset{\|\mathbf{a}_i\|_0}{\operatorname{argmin}} \mathbf{a}_i$
  subject to $\|\mathbf{y}_i - \mathbf{D}\mathbf{a}\|_2^2 \leq \epsilon$
  for $k = 1...K$ **do**
    $\mathbf{d} = \mathbf{Y}_I \alpha_k - \mathbf{D}\mathbf{A}_I \alpha_k$
    $\mathbf{d} = \frac{\mathbf{d}}{\|\mathbf{d}\|_2}$
    $\alpha = \mathbf{Y}_I^T \mathbf{d} - (\mathbf{D}\mathbf{A}_I)^T \mathbf{d}$
    $\mathbf{D}_k = \mathbf{d}$,
    $\mathbf{A}_I^k = \alpha^T$
  end for
end for

zero-mean, white, and homogeneous Gaussian noise n with standard deviation σ

$$y = x + n \tag{12}$$

in [11]. The K-SVD algorithm trains the dictionary on the image y in order to learn structure present in x while stationary Gaussian white noise is not learned.

Therefore, the image is decomposed into small patches of size $\sqrt{n} \times \sqrt{n}$ stored in vectors $\mathbf{y} \in \mathbb{R}^n$. The patches can be denoised by finding their sparse approximation for a given dictionary and then reconstructing them from the sparse representation. The denoised patch is obtained from $\hat{\mathbf{x}} = \mathbf{D}\hat{\mathbf{a}}$, after the sparse representation $\hat{\mathbf{a}}$ has been found by

$$\hat{\mathbf{a}} = \underset{\mathbf{a}}{\operatorname{argmin}} \|\mathbf{a}\|_0 \textit{ subject to } \|\mathbf{D}\mathbf{a} - \mathbf{y}\|_2^2 \leq T. \tag{13}$$

Here, T is dictated by ϵ and the standard deviation of noise σ of the patch. This optimization problem can also be reformulated, such that the constraint becomes a penalty

$$\hat{\mathbf{a}} = \underset{\mathbf{a}}{\operatorname{argmin}} \|\mathbf{D}\mathbf{a} - \mathbf{y}\|_2^2 + \mu\|\mathbf{a}\|_0. \tag{14}$$

The denoising K-SVD algorithm first trains the dictionary on patches of the noisy image $\mathbf{Y}$, and then reconstructs the denoised image $\mathbf{X}$ depending on the found dictionary. This can be formulated as an optimization problem [11]:

$$\hat{\mathbf{X}} = \underset{\mathbf{X},\mathbf{D},\mathbf{A}}{\operatorname{argmin}} \{\lambda\|\mathbf{Y} - \mathbf{X}\|_2^2 + \sum_p \mu_p \|\alpha_p\|_0 + \sum_p \|\mathbf{D}\mathbf{a}_p - R_p\mathbf{X}\|_2^2\} \tag{15}$$

Here, the parameter λ is a Lagrange multiplier that controls how close the denoised output image $\hat{\mathbf{X}}$ will be to the noisy image, as can be seen from the first term. The parameter μ_p determines the sparsity of the patch p. R_p denotes the matrix that extracts the patch p from the image. These patches are generally overlapping in order to avoid blocking artifacts at borders of patches.

In the K-SVD denoising algorithm, first the initial dictionary is defined, for example with atoms that contain signals of a discrete cosine transform (DCT), or with noisy patches from the image. The output image $\mathbf{X}$ is initialized with the noisy image, $\mathbf{X} = \mathbf{Y}$. Then, several iterations of the K-SVD algorithm are performed with the following steps:

- In the sparse coding stage, the sparse representation vectors $\mathbf{a}_p$ of each patch $R_p\mathbf{X}$ are computed. Here, any pursuit algorithm can be used to approximate the solution of

$$\forall_p \min\|\mathbf{a}_p\| \text{ s.th. } \|\mathbf{D}\mathbf{a}_p - R_p\mathbf{X}\|_2^2 \leq \left(C\sigma^2\right). \tag{16}$$

  The error tolerance is chosen to be the noise variance of the image multiplied by a gain factor C.

- The dictionary update stage (as described in section 2.2) is performed on the patches of the noisy image.

When the dictionary has been trained, the output image is computed by solving

$$\hat{\mathbf{X}} = \underset{\mathbf{x}}{\operatorname{argmin}}\{\lambda\|\mathbf{Y} - \mathbf{X}\|_2^2 + \sum_p \|\mathbf{D}\mathbf{a}_p - R_p\mathbf{X}\|_2^2\}. \tag{17}$$

The closed-form solution of this simple quadratic term is

$$\hat{\mathbf{X}} = \left(\lambda\mathbf{I} + \sum_p R_p^T R_p\right)^{-1}\left(\lambda\mathbf{Y} + \sum_p R_p^T\mathbf{D}\hat{\mathbf{a}}_p\right). \tag{18}$$

This solution can be computed by averaging the denoised patches, which are obtained from the coefficients $\mathbf{a}_p$ and the dictionary $\mathbf{D}$. Additionally, some relaxation is obtained by averaging with the original noisy image, dependent on the parameter λ.

This original K-SVD denoising algorithm from [11] works for homogeneous white Gaussian noise. But as described before, noise in CT images is nonstationary and can not assumed to be white and gaussian. Thus, the K-SVD denoising algorithm is adapted to work for CT images by means of considering the local noise variance, instead of assuming a constant global noise level, in order to be able to eliminate nonstationary white Gaussian noise.

Instead of a single input image, two images u_A and u_B with uncorrelated noise are considered to estimate the noise distribution. These two images are obtained by separate reconstructions from two disjoint subsets of projections with the same number of samples, $P1 \subset P$ and $P2 \subset P$, with $P1 \cap P2 = \emptyset$, $P1 \cup P2 = P$ [5]. The average image u_M is computed by $u_M = \frac{u_A + u_B}{2}$. For each patch of u_M that is considered in the algorithm, the noise variance is computed from corresponding pixels of the noisy difference image $u_D = u_A - u_B$. This value of the variance $V\left(R_p\mathbf{u_D}\right)$ for the corresponding patch (i, j) of the difference image is scaled in order to obtain the noise variance of the patch P of the considered average image

$$V\left(R_p u_M\right) = \frac{V\left(R_p\mathbf{u_D}\right)}{4} \tag{19}$$

The K-SVD denoising algorithm is decomposed into two stages:

- *Dictionary Training Stage:* Here, an initial dictionary D_{Init} is defined and then trained on randomly selected training patches of the noisy average image u_M by using the general K-SVD algorithm. The resulting dictionary D_{Final} represents structure of the image. After each dictionary update (except for the last) the dictionary is pruned from having too similar atoms, in order to ensure low mutual coherence (7) of the dictionary. These atoms are replaced by training patches with large relative representation error. The representation error is normalized by the error tolerance in order to ensure that no training patches with large representation error due to large amount of noise, are used for replacing atoms. In the same way, atoms are replaced that are used in sparse representations of too little training patches.

- *Image Denoising Stage:* The formerly trained dictionary is used for denoising overlapping patches of u_M and then computing the denoised output image u_{out} from a weighted average of the original noisy image u_M and the obtained image from the denoised overlapping patches u_{den}.

In order to control the sparse approximation error tolerance in addition to the local noise variance, we introduce gain factors that are multiplied by the local noise variance of each patch. In the dictionary training stage, this parameter is denoted by C_{Train}. It is used there as an additional means of controlling the proximity of noisy training patches and dictionary atoms. In the denoising stage, this parameter is denoted by C_{Den}. It steers the amount of noise in the denoised image. In the dictionary training stage of the CT denoising K-SVD algorithm, the error tolerance in the corresponding sparse coding stage of the original image denoising K-SVD algorithm – as shown in (16) – is replaced by

$$\forall_p \in \{TrPs\} \quad \min_{\mathbf{a}_p} \|\mathbf{a}_p\|_0 \quad s.th. \quad \|D\mathbf{a}_p - R_p u_M\| \leq (C_{Train} \cdot V(R_p u_M))) \,. \tag{20}$$

Consequently, for each training patch in the set $TrPs$, sparse representation coefficients $\mathbf{a}_p$ are computed up to the error tolerance controlled by C_{Train} and the local noise variance $V(R_p u_M)$ of the training patch. Likewise, the error tolerance in the sparse coding stage of the denoising stage with the trained dictionary D_{Final} is given by

$$\forall_p \quad \min_{\mathbf{a}_p} \|\mathbf{a}_p\|_0 \quad s.th. \quad \|\mathbf{D}_{Final}\mathbf{a}_p - R_p u_M\| \leq (C_{Den} \cdot V(R_p u_M))) \tag{21}$$

Since the error tolerances are now depenent on the noise variance of each patch, and not on a global noise estimate, the Lagrange multiplier λ in equation (17) becomes Λ_P, whose elements depend on the local noise variance of each patch, and the corresponding equation with the notation used in this section reads

$$u_{out} = \underset{u_{den}}{\operatorname{argmin}} \{\Lambda_P \|u_M - u_{den}\|_2^2 + \sum_p \|\mathbf{D}_{Final}\mathbf{a}_p - R_p u_{den}\|_2^2\}. \tag{22}$$

The output image u_{out} can be computed according to equation (18) for each pixel of the involved images by

$$u_{out} = \frac{u_M + \frac{1}{30}\frac{u_{supVar}}{u_{wght}} \cdot u_{supPt}}{1 + \frac{1}{30}\frac{u_{supVar}}{u_{wght}} \cdot u_{wght}} \tag{23}$$

This formula indicates which operations are applied to the corresponding pixels of each image. Here, u_{supPt} is the image that consists of the superimposed denoised patches, summed up at their corresponding positions. The image u_{wght} contains weights indicating in how many different overlapping patches a certain pixel was contained. The pixel-wise quotient of these two images results in the image u_{den} (which corresponds to $\mathbf{X}$ from equation (18)). The image u_{supVar} contains the superposition of the local noise variances in each patch. The factor $\frac{1}{30}$ corresponds to the recommendation in [11] to set $\lambda = \frac{30}{\sigma}$ for images in which a global noise estimate exists.

3 Implementation on Cell BE.

The Cell Broadband Engine (BE) is distinguished from current standard cache-based multicore architectures in that it contains two kinds of processors on a chip. One general purpose PowerPC Processor Element (PPE) for control-intensive tasks and eight co-processors called Synergistic Processor Elements (SPEs), 128-bit short-vector RISC processors specialized for

data-rich and compute-intensive SIMD applications. Instead of a cache the SPEs include local memory to store both instructions and data for the SPE. In that way, time demanding cache misses are avoided, but explicit DMA transfers are required to fetch data from shared main memory into the local store and back. This design explicitly parallelizes computation and transfer of data and instructions [14], since those transfers have to be scheduled by the programmer. In order to achieve maximum performance, the programmer has to exploit this kind of parallelism, together with data-level parallelism and thread-level parallelism. Data parallelism is available on PPE and particularly on SPEs in the form of vector processing, where in one clock cycle, a single instruction operates on multiple data elements (SIMD). Task parallelism is provided by the CBEA by means of software threads with one main thread on the PPE that creates sub-threads which control SPEs. After initialization, execution can proceed independently and in parallel on PPE and SPEs [12]. Depending on the most efficient work partitioning, each of the SPE threads can operate concurrently on different data and tasks. High performance in Cell programming can be achieved by making extensive use of intrinsics. Intrinsics are inline assembly-language instructions in form of C function calls that are defined in the SDK. By using these function calls, the programmer can control and exploit the underlying assembly instructions without directly managing registers. This is done by the compiler that generates code that uses these instructions and performs compiler-specific optimization tasks, like data loads and stores and instruction scheduling, in order to make the code efficient. SIMD registers are 128 bit wide and unified. Therefore, the same SIMD operations can be performed on any of the vector data types that have been defined in the SPE programming model for the C language. In this work, the single precision floating point arithmetic is used. This is due to the fact that the first generation of Cell processors, on which the algorithm has been programmed, is only fast for single precision. Hence, a SIMD vector contains four floating point values.

Sparse Coding is the most time involving part of the K-SVD denoising algorithm, even if the efficient Batch-OMP algorithm is used. This results mainly from the high number of overlapping patches that have to be denoised, and from many training patches required to train the dictionary properly. In order to speed up the algorithm, this part was chosen to be implemented in parallel on the Cell processor. Due to the fact that the number of coefficients is increasing in every iteration, an efficient vectorization of the complete Batch-OMP code is hard to achieve. This is the case for operations in the progressive Cholesky update. However, by restricting the size of patches to a multiple of 4, as well as the number of dictionary atoms K, the matrix-vector multiplications can be performed with maximal SIMD performance. These are used when computing the projection of the dictionary on the signal, as well as for updating the projection of the dictionary on the current residual. Furthermore, SIMD can then be fully exploited when searching the next atom with maximum correlation to the residual, as well as for computing the initial residual norm.

3.1 Dictionary Training

Training patches do not influence each other, and thus the coefficient computation can be done in parallel on different data.
In the parallelization strategy applied in this thesis, SPE threads are assigned with disjoint sets of patches and corresponding error tolerances, for which they perform sparse coding. Thus, work balancing is not an issue in terms of data dependencies.

However, it is not known a priori how long it takes for a certain SPE thread to perform sparse coding for its set of patches. This results from the fact that depending on the error tolerance and the dictionary, the number of coefficients to be found varies for different patches. Furthermore, the time a thread requires to start computation, to transfer patches

and coefficients, and to perform sparse coding, is unpredictable. This results from the fact that ressouces are shared between threads and that resources are managed by the operating system. Thus, it would be no good strategy to predefine the patches for which each thread has to compute coefficients. Instead, patches are distributed dynamically between threads in chunks of a given number of patches.

The data structure in main storage, in which sparse representation coefficients are stored after being computed by the SPUs, requires a special ordering of the coefficients. It would either take too long to let the PPU store the coefficients in this matrix while SPEs are running, or require additional communication. Thus, data structures to which SPEs can transfer their coefficients, are defined. When the SPEs are finished, the PPU stores coefficients from these data structures in the coefficient matrix data structure.

In order to distribute disjoint chunks of patches among the SPE threads, an atomic counter was used for synchronization. This counter indicates to SPE threads, for which patches coefficients have to be found next. It also defines the location, to which the corresponding cofficients have to be transferred. Each SPU program can compute the corresponding effective addresses from the start address of the data structures on the PPE.

Each SPU computes the coefficients for trainings patches. Since many SPEs run concurrently, the work has to be distributed among them. As already mentioned above, the work is dynamically distributed among the SPEs and the synchronization is done by an atomic counter.

In the upper part of figure 1, the instance TrainPatches of class PatchSet is shown, in which the patches and the corresponding error tolerances are stored in main memory. Training

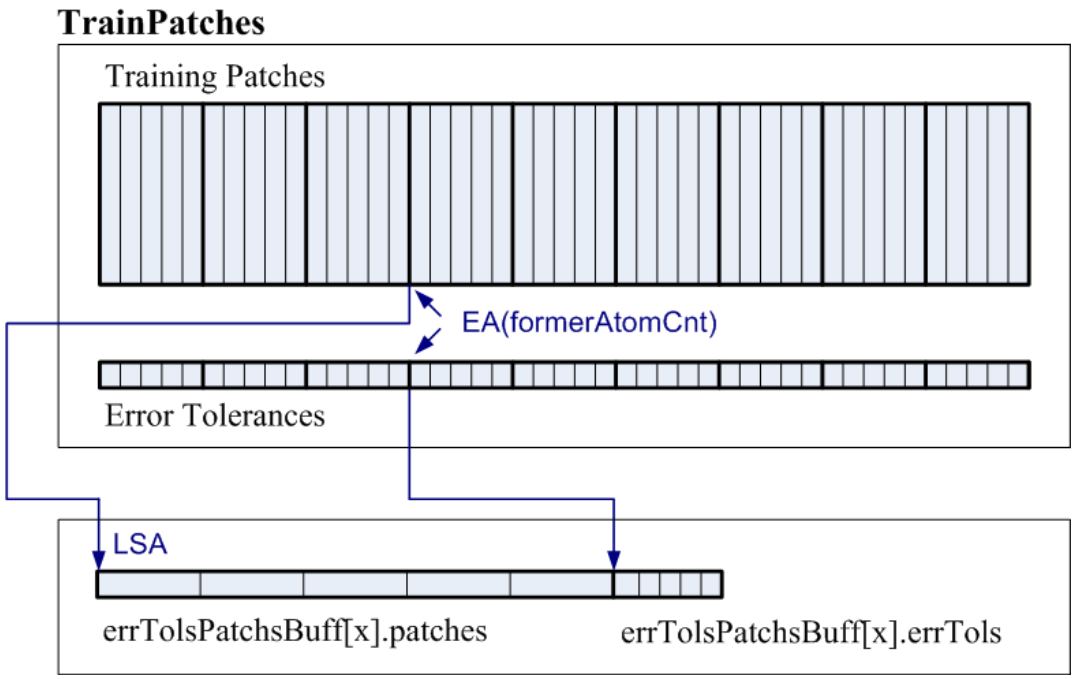


Figure 1: Work partitioning and DMA Transfer of Training Patches

patches are stored in an array that is denoted by TrainingPatches. The thin vertical lines represent borders of training patches, which are actually stored in row-major order. Several such training patches build a chunk of patches that is transferred to SPEs and for which coefficients are computed at once. These chunks are marked by bold lines. The corresponding error tolerances are stored in an array of floats in the same class.

A chunk of patches and corresponding error tolerances is transferred to one of the structures

in errTolsPatchsBuff on the SPE's LS, which is shown in the lower part of this figure.
Since for each patch a fixed number of coefficients `maxNumCoefsPATCH` has been reserved by the PPE, the address to which to transfer the coefficients, is uniquely defined.
Due to the double buffering that is described in the following, this offset is defined by the atomic counter value `oldAtomCnt` of the last data transfer.

In order to speed up the coefficient calculation, data transfer is overlapped with computation by means of double buffering. Before the SPU starts transferring data, it increments the atomic counter and stores the atomic counters' value before it was incremented. In case all other SPEs have already started computing coefficients of the last chunk of patches, the SPE indicates the PPE that it has nothing more to do by sending a mailbox message, and terminates.

The main part of the double buffering algorithm is shown below:

```
1  while (formerAtomCnt < ctx.numPatchPiles)
2  {
3    next_buffer = buffer^1; // switch buffer

5    // fetch patches and error tolerances to next buffer:
6    mfc_get((void *) &errTolsPatchsBuff[next_buffer], [...],
7                                tag_ids[next_buffer], 0, 0);
8    [...]

10   // wait for previously prefetched patches being transferred to LS
11   waittag(tag_ids[buffer]);

13   // compute coefficients of previously prefetched patches
14   compCoefs((coefStruct *)&computedCoefsBuff[buffer],
15             (errTolPatchStruct *)&errTolsPatchsBuff[buffer], [...]);

17   // transfer coefficients to PPE (main storage)
18   mfc_put((void *) &computedCoefsBuff[buffer].coefVals, [...],
19                                tag_ids[buffer],0,0);
20   [...]

22   // increase atomic counter and change corresponding variables
23   buffer = next_buffer;
24   oldAtomCnt = formerAtomCnt;
25   formerAtomCnt = _atomic_inc_return(synchronVars.cur_PatchPileCnt);
26 }
```

Inside the while loop, the buffer index that indicates to which buffer patches are transferred next, is switched. Then the DMA transfer to this buffer is initiated. The command in line 11 makes sure that patches and error tolerances have already been transferred to the former buffer. After that, the coefficient computation for patches in that former buffer is done by calling the corresponding function. This function stores the calulated coefficients for the given training patches and error tolerances in the buffer for coefficients with the same index from the last patch transfer. In line 18 the command can be seen for putting the computed coefficient values to the main storage. The same is done for the atom indices that correspond to the coefficients, and for the number of coefficients. After having initiated this data transfer to main memory, the buffer index of the patches that are currently transferred to the LS, is assigned to the buffer for which coefficients are computed in the next iteration. The same is done for the atom index.
In the last line, the atomic counter is incremented again. If it is smaller than the index of the last chunk of patches, the while loop is executed again. If this is not the case, i.e. another SPE has already reserved the last chunk, the following code is executed:

```
27  // wait for previously prefetched data
28  waittag(tag_ids[buffer]);

30  // compute coefficients of previously prefetched patches
31    compCoefs((coefStruct *)&computedCoefsBuff[buffer],
32              (errTolPatchStruct *)&errTolsPatchsBuff[buffer], [...]);

34  // transfer (last) coefficients to PPE (main storage)
35  mfc_put((void *) &computedCoefsBuff[buffer].coefVals, [...],
36                                tag_ids[buffer],0,0);
37  [...]
```

It first makes sure that all the DMA transfers to the buffer are completed. Then, the coefficient computation for these patches is performed and the coefficients are stored in their buffer. Finally, the DMA transfer of the coefficients to the main storage is started.
After that, the SPU waits until the coefficient transfer is completed, and then indicates the PPU that it finished its work.

3.2 Image Denoising

When computing denoised patches in the image denoising stage on a CPU, a method of the C++ K-SVD denoising framework can be used for extracting patches from the image. To let the PPU use this method and then send these patches to the SPEs would be very inefficient. Thus, patches have to be extracted by the SPEs from the noisy image stored in main storage. For that purpose, to each SPE thread a stripe of the noisy image is assigned, from which the SPU has to extract the patches. After having the patches at a certain position of the stripe extracted and denoised, the superposition of these patches is computed by the SPUs. This superposition is added to possibly formerly computed patch superpositions in the image at the current position, and then transferred back to the PPE.
The syncronization strategy is analgous to that of the dictionary training. Once the SPE thread finished denoising one stripe, it increases the atomic counter and denoises the next available stripe. When all stripes have been denoised, the threads are terminated by the PPE. If only one data structure for the superposition of the denoised patches was available on the PPE, SPEs would have to add their stripes to the stripes transferred before. This would require additional synchronization of SPEs' accesses to this data structure. In order to avoid this, separate data structures are provided by the PPE for each SPE.
The data structure `SupImpPatchsImgPPE` contains for each SPE thread a data structure of the size of an image, to which it can add its stripes of denoised patches. `WeightImgsPPE` provides a data structure of the same size, to which SPEs can add the number of patches in which each pixel was contained. These data structures are needed for computing the final denoised image by the PPE, after the SPEs have denoised all image stripes. First, the PPE sums up the images `SupImpPatchsImgPPE` and `WeightImgsPPE` from all SPEs. Then it computes the superposition of the noise variances for each patch, which will be needed by the C++ method that is invoked later for computing the final denoised image.

Double-buffering that was used for the dictionary training, was extended to quadro-buffering for the image denoising. This results from the fact that for extracting overlapping patches in horizontal and vertical direction, two of the previously described buffers are needed at once.
While at dictionary training a chunk of patches was assigned to each SPE, in the image denoising stage there is a horizontal stripe of the noisy image assigned to the SPEs. How these stripes are chosen from the image is shown in Figure 2. Which horizontal stripe is assigned, depends on the value of an atomic counter. Note that neighboring horizontal stripes are overlapping by `PATCHSIZE` elements.

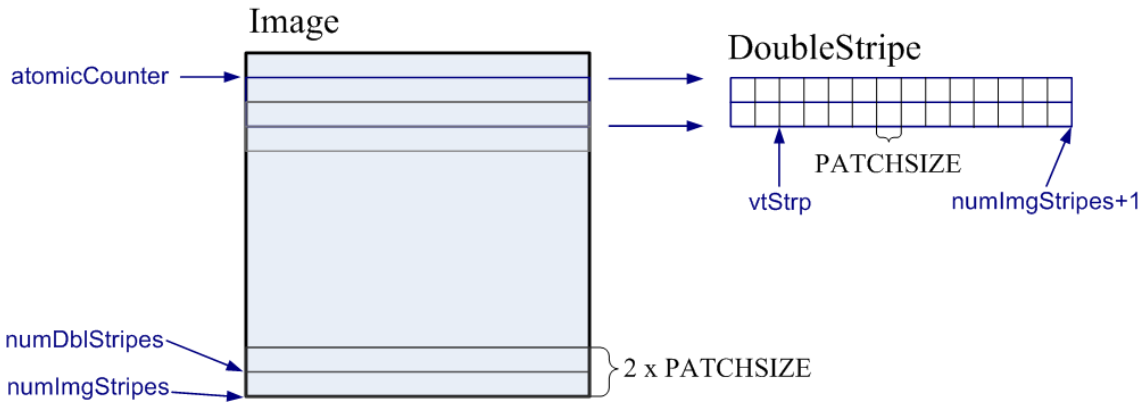


Figure 2: Work partitioning and DMA Transfer of Image Stripes

In order to be able to extract `PATCHSIZE` overlapping patches with distance of one pixel in vertical direction from the stripe, this stripe has a height of twice the patch-size. Thus, it is referred to as *double-stripe*.

Patches have to be extracted on the SPE from this double-stripe, since the data transfer is performed by means of DMA, which requires in general alignment at 16 byte boundaries in memory. Thus, blocks of the double-stripe with `PATCHSIZE` pixels in horizontal direction are transferred between LS and main storage. These blocks are shown in Figure 2 as vertical lines in the double-stripe.

Images in main storage are stored in row-major order. In order to let SPEs extract overlapping patches in vertical direction from the buffers in LS, `PATCHSIZE` pixel values from 2· `PATCHSIZE` rows are stored in buffers on the LS. This is shown in Figure 3.

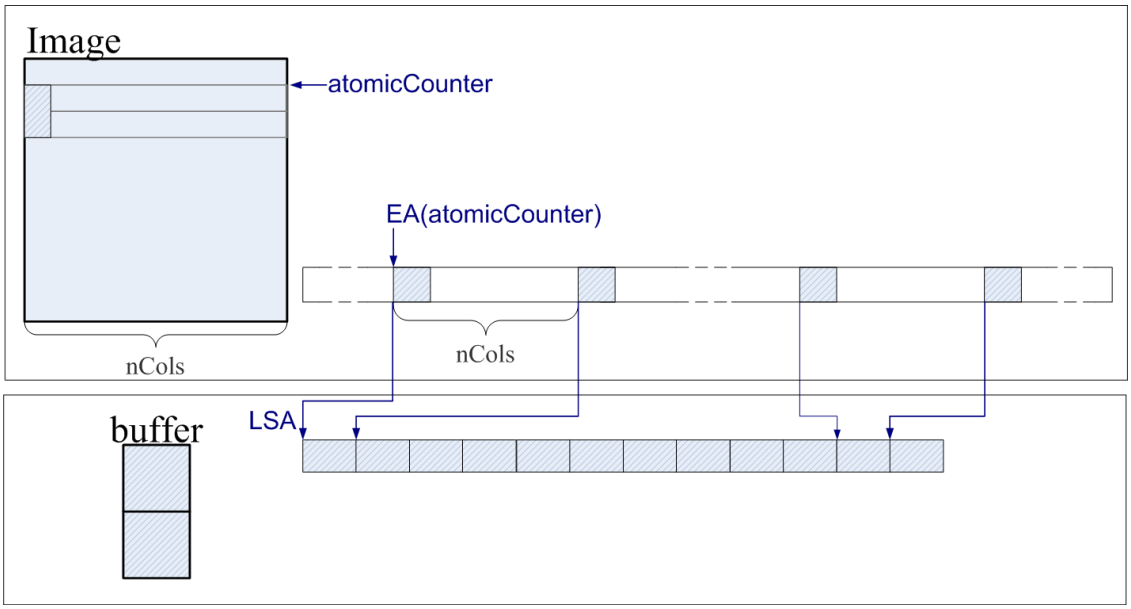


Figure 3: Transfer of image blocks by means of DMA-lists

Thus, several DMA transfers are needed to transfer data that stored in a discontinuous area in main storage to a continuous area in LS. These transfers can be managed by DMA-list transfers on the Cell. For transferring denoised patches back to the main storage, the same holds for the opposite direction.

For quadro buffering, four buffers at the local store (LS) are required for each of the following data: Noisy patches, denoised patches, weights in how many different patches a pixel was contained, and for the error tolerances.

Data in all of these buffers has to be transferred either from LS to main storage, or from main storage to the LS. In order to be able to wait for data transfers to complete before performing further steps, tag groups are built. For that purpose, four read - and four write tags are needed.

Like in the dictionary training algorithm, the SPU increments for the image denoising an atomic counter and is assigned with the corresponding double-stripe. In case the atomic counter is already larger than the number of stripes in the image, the SPE indicates the PPE that all double-stripes have already been distibuted among SPEs and terminates.

Then a loop over all double-stripes begins, and an inner loop over all blocks (vertical stripes) in the current double-stripe (see Figure 2). In case the current vertical stripe is not the last of those stripes in the double-stripe, the SPU waits for data with the current tag to be written to main storage. Then the DMA-list transfer from the next vertical stripe of the noisy image in the current double-stripe to a buffer for noisy image blocks in the LS, is initiated. The same is done for the denoised patches. They will be needed after the current patches have been denoised, and will be added to these already denoised patches in order to compute the superposition. The weights and the error tolerances are stored to their buffers, too. These buffers correspond to the current tag index (`vtStrp%numBUFFs`).
In case the current vertical stripe is the last one in the current double-stripe, no new data is transferred to the LS buffers.

After these DMA-list transfers have been initiated, the SPU waits for the DMA-list transfers that have been initiated in the previous iteration to finish. Of course only if the current iteration is not the first one.

If there are already two buffers available, the following instructions, shown as soucre code, are executed.

```
1  if(vtStrp >= 2) {
2    // determine the number of rows to be extracted from current double-patch
3    numPatchRowsExtract = (formerAtomCnt != (numDoublStripes-1)) ?
       PATCHSIZE : PATCHSIZE+1;

5    // compute denoised patches, add them to two most recent buffers
6    for(curRow=0; curRow<numPatchRowsExtract; ++curRow)
7    {
8      extractPatches((errTolPatchStruct *)&errTolsPatchsDen,        (vec_float4 *)
         noisyPatchsBuff[(vtStrp-2)%numBUFFs],        (vec_float4 *)
         noisyPatchsBuff[(vtStrp-1)%numBUFFs],          curRow);
9      extractErrTols((errTolPatchStruct *)&errTolsPatchsDen,       (vec_float4 *)
         errTolsBuff[(vtStrp-2)%numBUFFs], curRow);
10     compCoefs(...&computedCoefs, ...&errTolsPatchsDen, [...]);
11     reconstrAddPatchs((vec_float4 *)densdPatchsBuff[(vtStrp-2)%numBUFFs],
                                                 (vec_float4 *)densdPatchsBuff[(
         vtStrp-1)%numBUFFs],        (coefStruct *)&computedCoefs, (vec_float4 *)
         Dict, curRow);
12     addWeights((vec_uint4 *)WeightsBuff[(vtStrp-2)%numBUFFs],              (
         vec_uint4 *)WeightsBuff[(vtStrp-1)%numBUFFs],            curRow);
13   } // end for
```

First, the number of rows of the current buffers, for which the following computations will be done in the following for-loop, is determined in line 3. The loop iterates over the rows of the two current buffers for noisy patches, denoised patches, error tolerances and weights.

In the current row of both buffers that contain noisy image blocks, patches are extracted. These are stored in the input data structure for the Batch-OMP algorithm. The first buffer in the function call is the buffer, from which patches have been extracted in the previous iteration. This buffer corresponds to buffer2 in Figure 4. The content of second buffer has just been transferred to the LS.
The next function that is invoked extracts error tolerances for sparse coding from its buffer and stores them in the input data structure for the Batch-OMP algorithm. These error tolerances correspond to the previously extracted patches.
Then the coefficients of the extracted noisy patches are computed. These coefficients are used as input parameters for the next function.
This function reconstructs the noise-free patches from the coefficients and the dictionary. The reconstructed patches are then shifted to the position from which they have previouly been extracted, and are summed up with the previously denoised patches in the corresponding buffers. Finally, the weights that correspond to the denoised patches are summed up with previously computed weights in their buffers.

The reason for making the for-loop termination criterion dependent on the current doublestripe is that usually from `PATCHSIZE` rows of the doublebuffer patches are extracted. The last row is already the first patch of the next doublestripe. However, when the last doublestripe is denoised, there is no next one. Thus, the last row of patches is also denoised in this case.

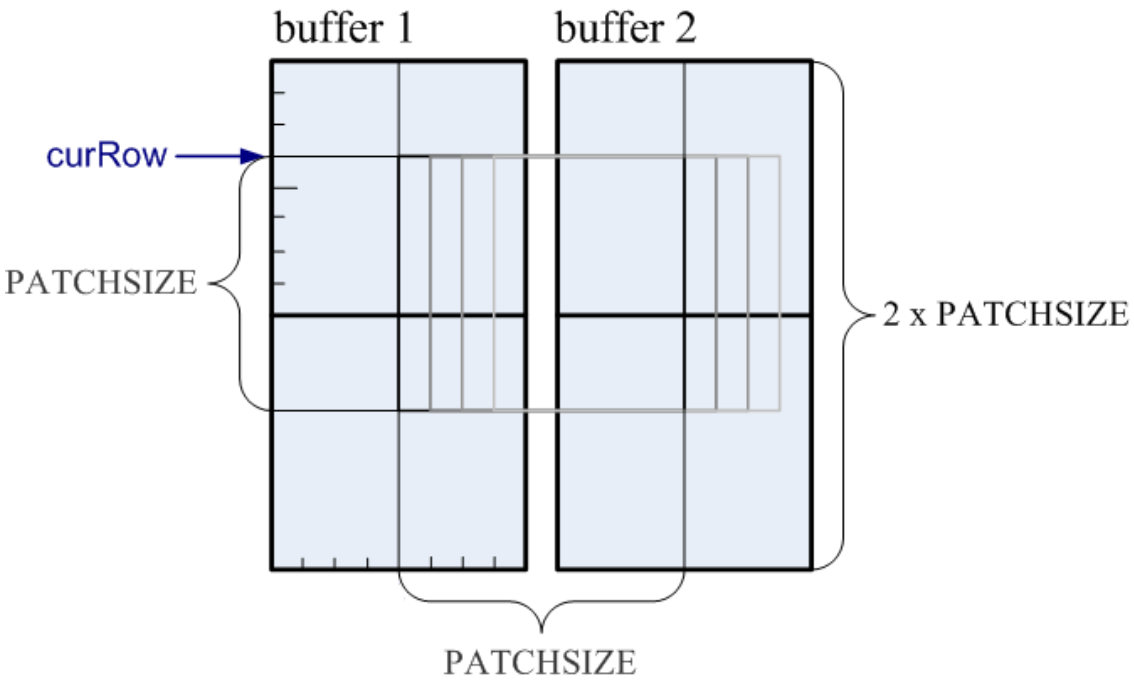


Figure 4: Extraction of overlapping patches from two buffers

Now that the patches in all rows of the buffers have been denoised, the buffer with denoised patches and the buffer with the weights, to which patches have been added twice, are transferred to main storage.

```
14    putPatchsPPE((void* )densdPatchsBuff[(vtStrp-2)%numBUFFs],
          SupImpPatchsImgPPE_ptr,                    cmpOffset_Patchs(formerAtomCnt,
          vtStrp-2), (vtStrp-2)%numBUFFs, write_tags[(vtStrp-2)%numBUFFs]);
15    putWeightsPPE((void* )WeightsBuff[(vtStrp-2)%numBUFFs], ...);
16  } // end if
```

Here, all vertical stripes of the current double-stripe, except for the last one, have been denoised.

This last vertical stripe requires some special treatment. Like for the last double-stripe described before, the last vertical stripe has no neighbour. Thus, the last patches in the column also have to be denoised.

A last special case is the patch in the right lower corner of the image. For this case all functions for extracting patches and tolerances, sparse coding and reconstruction of denoised patches, and more are performed in a single function.

Since the current doublestripe has now completely been denoised, the atomic counter is increased in order to get the next double-stripe assigned. Before that, the content of the buffer to which denoised patches have been added twice, and of the corresponding weights buffer are transferred to main storage.

At the end of this algorithm, the PPE is indicated by means of a mailbox message that the computations are finished, and the SPU program terminates after having freed all allocated memory and released all tag IDs.

4 Results

4.1 Computational Performance

Next we measure the performance of parts of the K-SVD algorithm. The time measurement was performed by means of `gettimeofday()` C function.

Comparison of Runtime on Cell and CPU: First, the runtime of the Cell implementation on a PlayStation® 3 with 6 SPEs was compared to the corresponding C++ implementation on an Intel® Core™2 Duo CPU 6400 with 2.13GHz. The following parameters were used:

- The number of coefficients was set to 16 and the parameters C_{Train} and C_{Den} were set to zero.
  This way, for each patch there is the same number of coefficients found, independently of the local noise variance
- The number of training patches was set to 8000, since the original K-SVD algorithm takes too long otherwise
- The number of dictionary atoms was set to 128, the patch size to 8
- The image that was denoised had a size of 512 x 512 pixel

Execution times are shown in table 1. We show times for sparse coding for computing the sparse representation coefficients of training patches in the dictionary training stage (Train) and the computation of the sparse representation of image patches and the superposition of denoised patches in the image denoising stage (Denoise). In addition, the dictionary update (Dict Update) and the dictionary pruning step (Clear Dict) are compared to each other. The corresponding methods are implemented only in C++. Thus, these methods were only run by the PPE on the Cell.

Table 1: Execution times on CPU and Cell in seconds. CPU 1 uses K-SVD and OMP, CPU 2 and Cell approximate K-SVD and Batch-OMP.

Method	Train	Dict Update	Clear Dict	Denoise
CPU 1	21.10	374.10	0.02	700.00
CPU 2	1.55	0.69	0.02	51.00
Cell	0.10	1.75	0.11	3.10

Scaling on the Cell: Scaling results were produced on the PlayStation® 3 and the Cell Blade QS20 and for the same setting and parameters as in the previous experiment.

Strong scaling was measured for the sparse coding in the dictionary training stage by fixing the problem size to 24000 training patches and increasing the number of SPEs for computations. Results for dictionary training and image denoising are shown in figure 5. Measurements on PS3 and on the Cell Blade conincide. As expected with increasing number

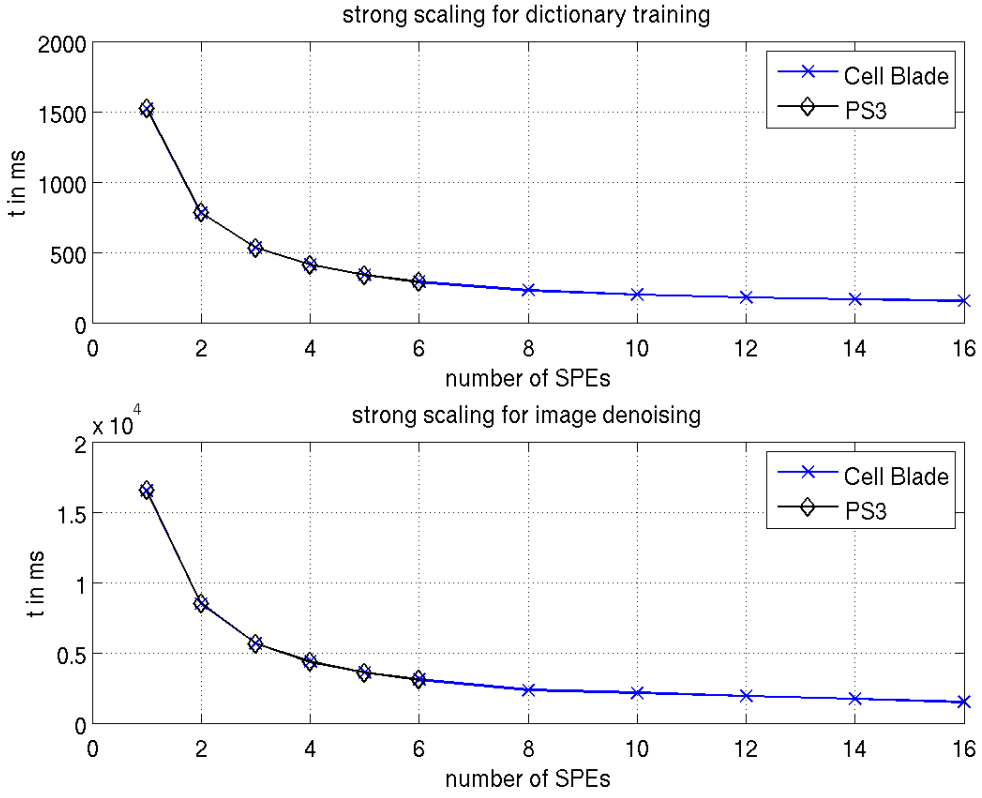


Figure 5: Strong scaling of K-SVD on Cell.

of SPEs the speedup is decreasing.

The parameters for weak scaling on the PS3 are summarized in table 2. Here, patches

Table 2: Weak scaling parameters.

# SPEs	1	2	4	6
# patches	3000	6000	12000	18000
# pixel	208	296	416	512

denotes the number of used training patches, and # pixel the number of pixel in each dimension of the image. All other parameters had the same values as in the strong scaling measurements. In figure 6, a graph of the weak scaling results is found. Here, the execution time stays almost constant with increasing number of SPEs and thus weak scaling is nearly optimal.

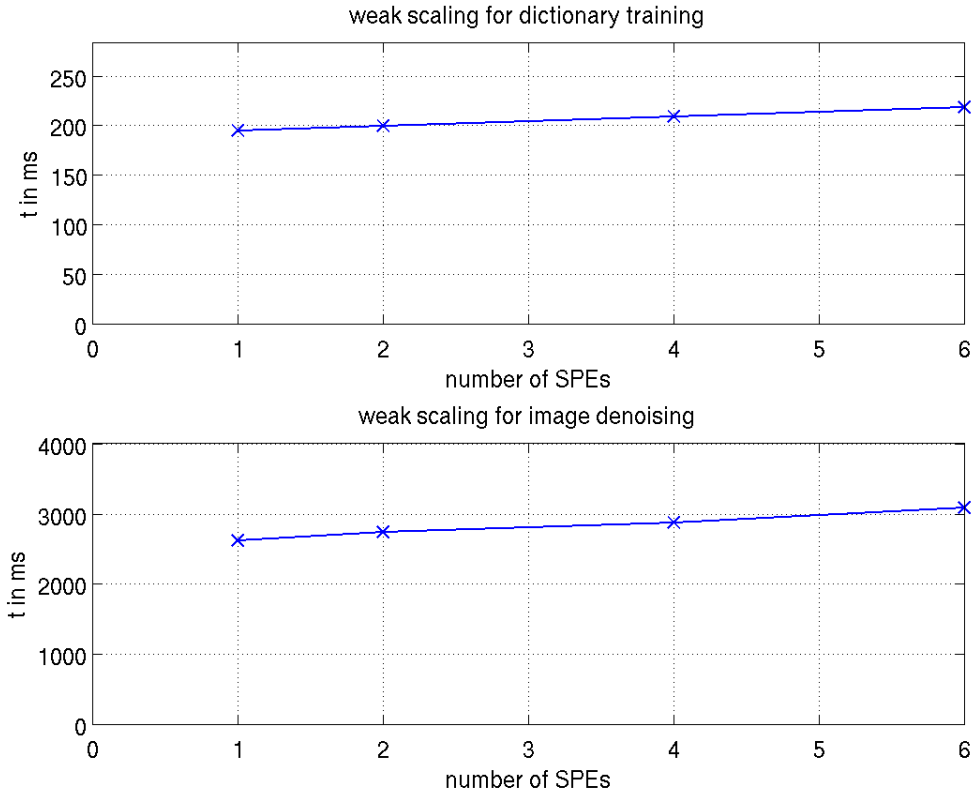


Figure 6: Weak scaling of K-SVD on Cell.

4.2 Quantitative Denoising Results

In order to quantify how much noise is suppressed by a denoising method, the noise reduction rate (NRR) is computed. A measure for how well structure is preserved is obtained from the modulation transfer function (MTF). The MTF was determined at an edge of the phantom images generated by the *DRASIM* software package [1]. In addition to the object data, also quantum noise was simulated in these images and noise-free ground truth data is available for computing noise reduction. In figure 7, we marked the region of a phantom image in which the MTF was measured. As described in [5], the edge profile was sampled in the selected region around an edge, and then this profile was averaged along the edge. By deriving the edge profile, the line spread function (LSF) is obtained. The Fourier transform of the LSF leads to the optical transfer function (OTF), whose magnitude is the MTF. The MTF is normalized to $MTF(0) = 1$. For comparing different MTFs, the MTF(ρ_{50}) = 0.5 value can be used. It represents the spatial frequency for which an MTF has a value of 0.5. From the ρ_{50} value the edge-preservation rate (EPR) is computed [5]

$$EPR = \frac{\rho_{50}^{denoised}}{\rho_{50}^{original}}. \tag{24}$$

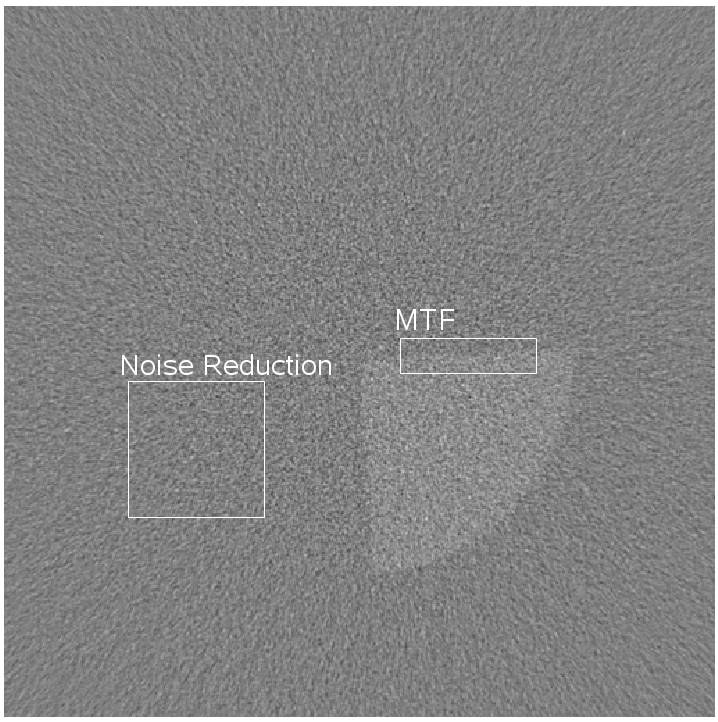


Figure 7: Example of noisy phantom image with 10HU contrast for measuring NRR and MTF (adapted from [5]).

It describes how well edges are preserved, by comparing the ρ_{50} value of a denoised image with the (ρ_{50}) value of an ideal noise-free image. The NRR is now defined as [5]

$$NRR = 1 - \frac{\sigma^{denoised}}{\sigma^{noisy}}. \tag{25}$$

It is obtained by measuring the standard deviation of the noise in a homogeneous region of the phantom images (see figure 7).

Reliable measurements of the MTF with the edge method described above can only be achieved, if the contrast of the edge is much higher than the noise at the edge [7]. Since this is not the case in the phantom images, for each MTF measurement for a given contrast, the MTF was computed from the average of 100 denoised slices. The noise reduction rate was also obtained by computing the standard deviation of the noise in these 100 slices and then computing the mean value of these standard deviations. Note that when the K-SVD algorithm is used for training a dictionary on these noisy phantoms, the dictionary will mainly learn noise, if anything at all. This results from the fact that very little structure is present in these images, since the phantoms consist mainly of homogeneous regions. Thus, the denoising was performed with an untrained undercomplete DCT dictionary with 36 atoms, and with a dictionary with 128 atoms, which has been trained before on Abdomen images for $C_{Train} = 30$ with 5 K-SVD iterations.

MTFs for phantom images of different contrasts that have been denoised with the elsewhere trained dictionary are shown in figure 8 for $C_{Den} = 30$. In order to see how an ideal MTF would look like for these images, an MTF is shown, which has been computed from the ideal noise-free phantom image with 100HU contrast. This MTF is denoted by *100HU orig* in the MTF plot. The closer other MTFs are to this curve, the better the denoising result. As expected, MTFs for smaller contrasts fall further below the ideal curve than MTFs of images with higher contrasts. That means, the smaller the contrast is, the more resolution

is lost. However, edges are preserved well with this method for the given parameters, since all MTFs are close to the ideal curve. The corresponding edge preservation rate and the noise reduction rate are shown in figure 9. For the EPR, the reference images are for each contrast the corresponding ideal phantom images. As already seen from the MTFs, the EPR decreases for images with smaller contrasts. The NRR plot shows that the noise is reduced by approximately 50% for all contrasts.

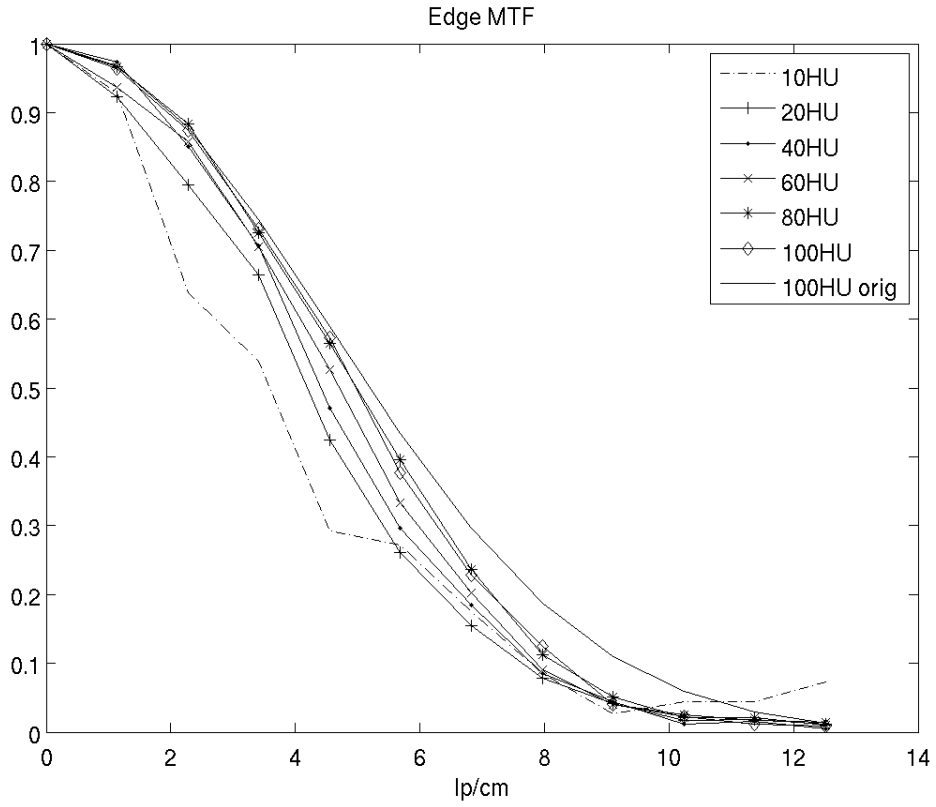


Figure 8: MTFs of phantom images with different contrasts denoised with K-SVD for $C_{Train} = 30$ and $C_{Den} = 20$.

In order to get an overview of the relationship between EPR and NRR for different values of the parameter C_{Den}, EPR and NRR are shown in figure 10 for $C_{Den} \in [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 75, 100]$. The values of EPR and NRR were computed from denoised phantom images with 100HU contrast. The plot contains one curve for the untrained DCT dictionary, and one curve for the dictionary that was trained on a real medical image (see figure 11). Measurement points with a high NRR correspond to large values of C_{Den}, those points with a small NRR correspond to small values of that parameter. As expected, the EPR rises with falling NRR, i.e. the more noise is reduced, the stronger edges are blurred. The EPR for a given NRR is much better for the trained dictionary than for the untrained one, even though the dictionary was trained on a different CT image.

4.3 Qualitative Denoising Results

In order to get an impression of the visual quality of the CT denoising K-SVD algorithm and to find parameters for which is works best, we tested it on real clinical CT data sets. A slice of one of these of the upper body and mainly containing the liver is shown in figure 11. This image has been obtained by averaging the two input images that have been reconstructed

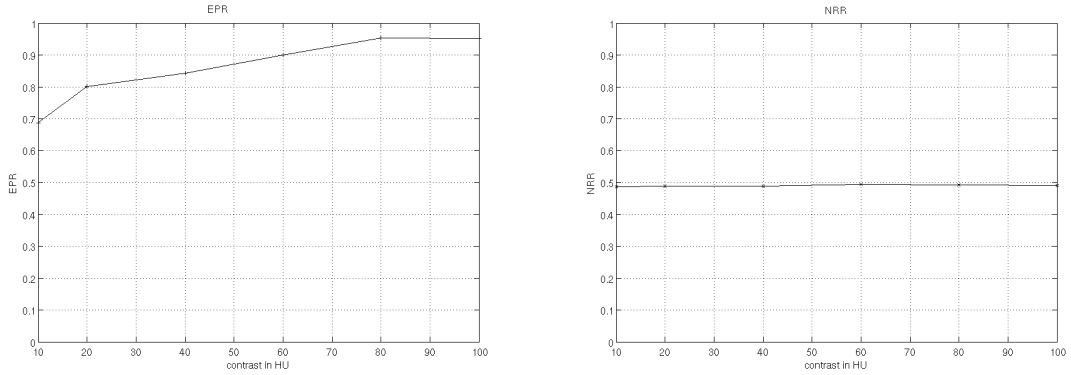


Figure 9: EPR (left) and NRR (right) for phantom images with different contrasts denoised with K-SVD for $C_{Train} = 30$ and $C_{Den} = 20$.

from disjoint projections (the odd and the even ones) of the same scan. This average image is equal to the image that would be obtained when all projections are used for reconstructing the image. The difference image u_D – computed by $u_D = \frac{u_A - u_B}{2}$ for input images u_A and u_B – visualizing uncorrelated noise contained in both reconstructed images is depicted in figure 11. Since gray values in a reconstructed CT image can be distributed over different ranges of houndsfield units (HUs), depending on the tissue and on scanning parameters, windowing is performed for displaying the images. That means only HUs in a certain range, or window, are represented as gray valued in the image. Parameters for windowing are c, the HU at the center of the window, and w, the width of that window. The Liver image is displayed for $c = 200$ and $w = 700$, whereas the difference images are displayed for $c = 0$ and $w = 200$.

In figure 12 denoised Liver images are found for different parameters and $C_{Train} = 30$ together with the corresponding difference images. Best results in terms of noise reduction and edge preservation seem to be achieved for $C_{Den} = 20$. For lower values, the edges are well preserved, but the noise reduction decreases. For higher values edges are blurred too much.

5 Conclusion and Outlook

In this paper, a patch-based algorithm for denoising of images was adapted to medical CT images.

Our first image denoising results are promising. A qualitative analysis was performed showing that the K-SVD CT denoising algorithm considers the nonstationarity of noise. Furthermore, structures and edges are preserved, while noise is reduced. A quantitative analysis compared noise reduction and edge preservation dependent on the contrast of denoised images. Here it is found that dictionary training is efficient, i.e. denosing by means of a dictionary that was trained on other CT images delivered better results than an untrained DCT dictionary.

The implemented algorithm provides a solid basis for further investigations in several directions. These include incorporation of the covariance into the sparse coding, in order to reduce non-Gaussian noise more effectively. Additional statistical parameters like the correlation coefficient could be incorporated into the K-SVD CT denoising. Furthermore, the existing algorithm for 2D could be extended to 3D, which was shown to lead to further improvements of the denoising properties for similar algorithms. In addition to the currently implemented sparse coding for dictionary training and image denoising, also the dictionary update could be migrated to the Cell. Another direction worthwhile to go is the implementation of the sparse coding algorithm and the dictionary update on Graphics Cards.

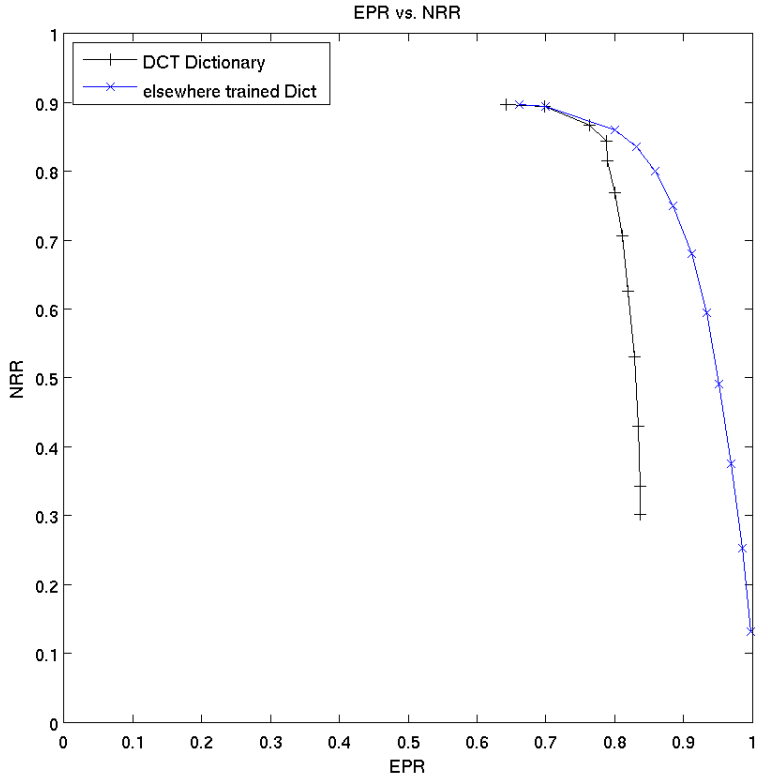


Figure 10: Relationship between EPR and NRR for different values of C_{Den} measured at phantom image with 100HU contrast for an untrained DCT dictionary and a trained dictionary.

References

[1] Segmented multiple plane reconstruction: A novel approximate reconstruction scheme for multi-slice spiral CT.

[2] M. Aharon, M. Elad, and AM Bruckstein. The K-SVD: An algorithm for designing of overcomplete dictionaries for sparse representation. *IEEE Transactions on Signal Processing*, 54(11):4311–4322, 2006.

[3] A. Borsdorf, R. Raupach, and J. Hornegger. Wavelet based Noise Reduction by Identification of Correlation. In K. Franke, K. Müller, B. Nickolay, and R. Schäfer, editors, *Pattern Recognition (DAGM 2006), Lecture Notes in Computer Science*, volume 4174, pages 21–30, Berlin, 2006. Springer.

[4] A. Borsdorf, R. Raupach, and J. Hornegger. Separate CT-Reconstruction for Orientation and Position Adaptive Wavelet Denoising. In A. Horsch, T. Deserno, H. Handels, H. Meinzer, and T. Tolxdoff, editors, *Bildverarbeitung für die Medizin 2007*, pages 232–236, Berlin, 2007. Springer.

[5] Anja Borsdorf, Rainer Raupach, and Joachim Hornegger. Separate CT-Reconstruction for 3D Wavelet Based Noise Reduction Using Correlation Analysis. In Bo Yu, editor, *IEEE NSS/MIC Conference Record*, pages 2633–2638, 2007.

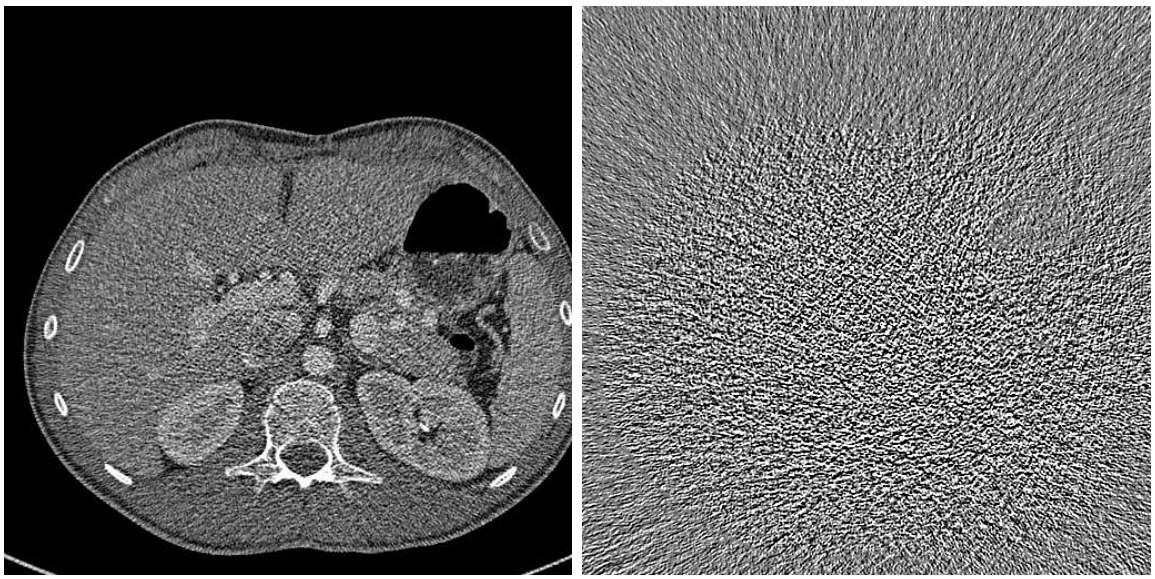


Figure 11: Liver CT scan average image (left) and noise image (right).

[6] SG Chang, B. Yu, and M. Vetterli. Spatially adaptive wavelet thresholding with context modeling forimage denoising. *Image Processing, IEEE Transactions on*, 9(9):1522–1531, 2000.

[7] I. A. Cunningham and B. K. Reid. Signal and noise in modulation transfer function determinations using the slit, wire, and edge techniques. *Medical Physics*, 19(4):1037–1044, July 1992.

[8] G. DAVIS, S. MALLAT, and M. AVELLANEDA. Adaptive greedy approximations. *Constructive approximation*, 13(1):57–98, 1997.

[9] D.L. Donoho. Denoising by soft-thresholding. *IEEE Trans. Inform. Theory*, 41(3):613–627, 1995.

[10] DL Donoho, M. Elad, and VN Temlyakov. Stable recovery of sparse overcomplete representations in the presence of noise. *Information Theory, IEEE Transactions on*, 52(1):6–18, 2006.

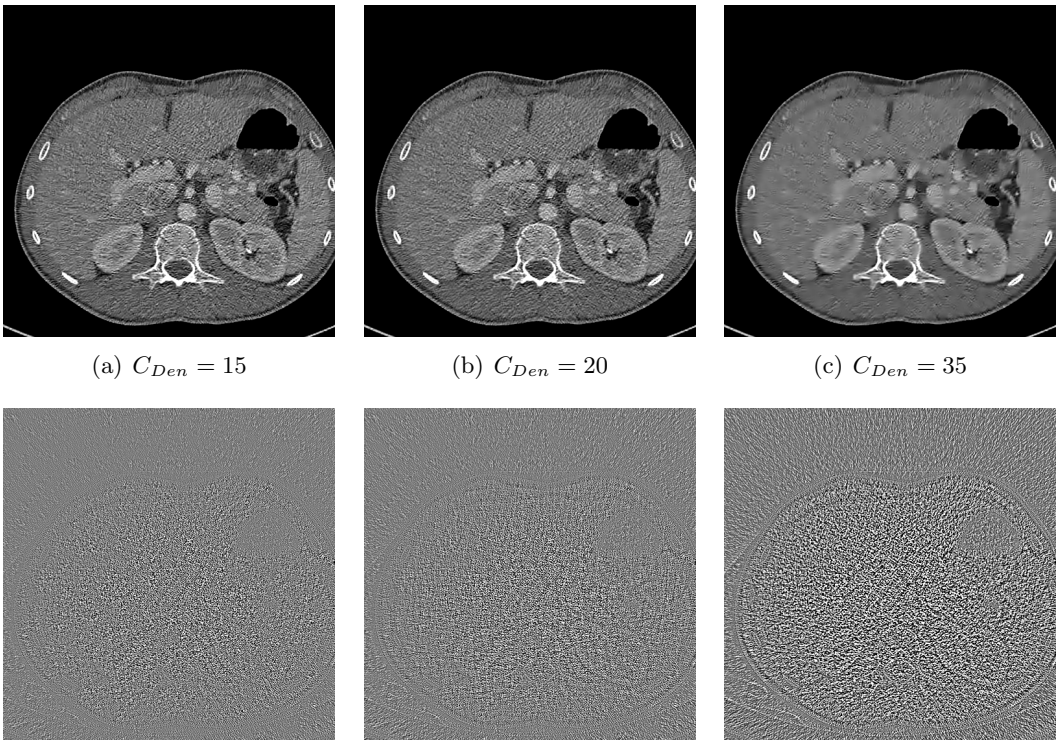
[11] M. Elad and M. Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *IEEE Trans. Image Process*, 15(12):3736–3745, 2006.

[12] M. Gschwind. The Cell Broadband Engine: Exploiting Multiple Levels of Parallelism in a Chip Multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[13] J. Hsieh. Adaptive streak artifact reduction in computed tomography resulting from excessive x-ray photon noise. *Medical Physics*, 25:2139, 1998.

[14] IBM Corporation, Rochester MN, USA. *Programming Tutorial, Software Development Kit for Multicore Acceleration, Version 3.0*, 2007.

[15] A. C. Kak and Malcolm Slaney. *Principles of Computerized Tomographic Imaging*. IEEE Press, 1988.



(a) $C_{Den} = 15$ (b) $C_{Den} = 20$ (c) $C_{Den} = 35$

(d) $C_{Den} = 15$ (e) $C_{Den} = 20$ (f) $C_{Den} = 35$

Figure 12: Denoised Liver CT scans ($C_{Train} = 30$).

[16] P.J. La Riviere, Junguo Bian, and P.A. Vargas. Penalized-likelihood sinogram restoration for computed tomography. *Medical Imaging, IEEE Transactions on*, 25(8):1022–1036, Aug. 2006.

[17] Tianfang Li, Xiang Li, Jing Wang, Junhai Wen, Hongbing Lu, Jiang Hsieh, and Zhengrong Liang. Nonlinear sinogram smoothing for low-dose x-ray ct. *Nuclear Science, IEEE Transactions on*, 51(5):2505–2513, Oct. 2004.

[18] M. Mayer, A. Borsdorf, H. Köstler, J. Hornegger, and U. Rüde. Nonlinear Diffusion Noise Reduction in CT Using Correlation Analysis. In J. Hornegger, E. Mayr, S. Schookin, H. Feußner, N. Navab, Y. Gulyaev, K. Höller, and V. Ganzha, editors, *3rd Russian-Bavarian Conference on Biomedical Engineering*, volume 1, pages 155–159, Erlangen, Germany, 2007. Union aktuell.

[19] M. Mayer, A. Borsdorf, H. Köstler, J. Hornegger, and U. Rüde. Nonlinear Diffusion vs. Wavelet Based Noise Reduction in CT Using Correlation Analysis. In H.P.A. Lensch, B. Rosenhahn, H.-P. Seidel, P. Slusallek, and J. Weickert, editors, *Vision, Modeling, and Visualization 2007*, pages 223–232, 2007.

[20] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(7):629–639, 1990.

[21] R. Rubinstein, M. Zibulevsky, and M. Elad. Efficient Implementation of the K-SVD Algorithm and the Batch-OMP Method.

[22] Joachim Weickert. Theoretical foundations of anisotropic diffusion in image processing. In *Theoretical Foundations of Computer Vision*, pages 221–236, 1994.