GPU PARALLEL IMPLEMENTATION OF THE APPROXIMATE K-SVD ALGORITHM USING OPENCL

Paul Irofti[*], *Bogdan Dumitrescu*[*†]

[*] Department of Automatic Control and Computers
University Politehnica of Bucharest
313 Spl. Independenţei, 060042 Bucharest, Romania

[†] Department of Signal Processing
Tampere University of Technology
PO BOX 553, 33101, Tampere, Finland

ABSTRACT

Training dictionaries for sparse representations is a time consuming task, due to the large size of the data involved and to the complexity of the training algorithms. We investigate a parallel version of the approximate K-SVD algorithm, where multiple atoms are updated simultaneously, and implement it using OpenCL, for execution on graphics processing units (GPU). This not only allows reducing the execution time with respect to the standard sequential version, but also gives dictionaries with which the training data are better approximated. We present numerical evidence supporting this somewhat surprising conclusion and discuss in detail several implementation choices and difficulties.

Index Terms— sparse representation, dictionary design, parallel algorithm, GPU, OpenCL

1. INTRODUCTION

OpenCL [1] is an open standard allowing portable parallel programming, aimed especially at graphics processing units (GPU) but not restrained to them. Despite its recent proposal, OpenCL has gained support from the industry and its implementation is supported by the major GPU manufacturers. Although some implementations miss certain features and there are difficulties in portability [2], there are much more incentives for using OpenCL than languages specialized to a single type of GPUs.

Signal processing has been a field of active development for GPU algorithms and, in the last few years, for OpenCL implementations. The problems solved in this framework are typical for image or video processing, as naturally fit for GPUs. There is work on segmentation [3], feature matching [4], motion estimation [5] and real time particle filtering [6], among others. Also more intensive computation tasks have been tackled, like the solution of optimization problems (rank minimization) [7]. Closer to our interest, we see algorithms for computing sparse representations [8].

In this paper, we focus on the problem of designing dictionaries for sparse representations [9], which is as follows. Given a set of m signals (or patches) $Y \in \mathbb{R}^{p\times m}$ and a target sparsity level s, find dictionary $D \in \mathbb{R}^{p\times n}$ (whose columns are named atoms) such that $\|E\|_F$ is minimized, where

$$E = Y - DX \tag{1}$$

is the approximation error and $X \in \mathbb{R}^{n\times m}$ is the matrix of sparse representations, having at most s nonzero elements on each column. Denoting Y_j a column of Y and X_{ij} an element of X, that means that

$$Y_j \approx DX_j = \sum_{i\in\mathcal{I}} D_i X_{ij}, \;\; |\mathcal{I}| = s, \tag{2}$$

i.e. a signal vector Y_j is approximated using only s atoms.

The problem is difficult since both the dictionary D and the sparse representation matrix X are unknown. The most successful algorithms like MOD [10] and K-SVD [11] alternate the optimization of the unknowns. Given an initial dictionary D, chosen randomly or as a selection of signal vectors, the sparse representations (2) are found; Orthogonal Matching Pursuit (OMP) is often preferred due to its speed. Then, keeping X fixed, a new dictionary is designed such that $\|Y - DX\|_F$ is minimized or reduced. K-SVD optimizes the atoms one by one, in the process changing also the representations corresponding to that atom. An SVD decomposition is used to this purpose, for each atom. To reduce the computation time, the Approximate K-SVD (AK-SVD) algorithm [12] replaces the SVD decomposition with a single iteration of the power method. Further improvements are suggested in [13, 14].

Our contribution is a parallel version of AK-SVD and its full implementation in OpenCL. Parallelism is present at several levels in the algorithm, but the atom update process is essentially sequential. The reason is that a modification of an atom is immediately taken into account for the next updates, in the general style of Gauss-Seidel iterations. We investigate here a version of the algorithm where more atoms are updated

This work was supported by the Romanian National Authority for Scientific Research, CNCS - UEFISCDI, project number PN-II-ID-PCE-2011-3-0400. P. Irofti was also supported by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/132395. E-mails: paul@irofti.net, bogdan.dumitrescu@tut.fi.

simultaneously. This not only takes less time on the GPU, but, surprisingly, may lead to smaller approximation error in (1).

2. PARALLEL AK-SVD WITH OPENCL

The main operations of our algorithm are shown in Algorithm 1. Steps 2–6 describe the first stage of AK-SVD, the computation of sparse representations, and steps 7–14 describe the second stage, the atoms update. We will present first the main ideas behind our parallel version, then give some details on the OpenCL implementation.

2.1. Parallel signal representations

The first stage of the AK-SVD algorithm is naturally parallel, since the signal representations (2) are completely independent. We first precompute the scalar products between the atoms and themselves and between the atoms and the signal vectors (steps 2–3). Since these are matrix multiplications of fairly large size, they can be easily parallelized. To compute the sparse representation of a signal, we use the Batch OMP (BOMP) algorithm, using the same operations as in the AK-SVD algorithm [12]; this algorithm builds the Cholesky decomposition of the matrix of the normal system associated with the sparse least-squares problem; the selection of columns is made by the standard matching pursuit criterion.

We compute the sparse representations in parallel for groups of $\tilde{m}$ signals. This is a compromise between leaving full decision to the GPU scheduler (when $\tilde{m} = m$) and a tight control of the parallelism (when $\tilde{m}$ is small, for example equal to the number of compute units). However, we did not notice significant differences between values from 1024 to m.

2.2. Parallel atoms update

The operations in the inner loop (steps 11–14) update an atom D_j and its coefficients in the representations where it appears. These operations are a direct implementation of the power method applied to the matrix $F^T F$, where F is the error matrix (1) from which the contribution of the current atom is removed. The signals whose current representations do not contain the current atom are not involved in the computation, hence the matrix F has a smaller number of columns ($|\mathcal{I}|$ instead of m). Note that F is not explicitly computed, but its expression is inserted in steps 13 and 14 and the computations proceed accordingly.

We introduce a new parameter $\tilde{n}$, which is the number of atoms processed in parallel with the power method. This is a departure from the standard AK-SVD algorithm, where $\tilde{n} = 1$. An obvious advantage is the parallelism, the maximum being obtained when $\tilde{n} = n$; this corresponds to a Jacobi variant of the K-SVD algorithm, opposed to the usual Gauss-Seidel form for $\tilde{n} = 1$. A possible drawback is a slower convergence, since the atoms from the same group of size $\tilde{n}$ cannot influence each other and hence further reduce the representation error.

In order to make the parallel updates possible, we compute in step 9 the current error matrix, which is hence updated after every $\tilde{n}$ atom updates. This is a significant computation effort if $\tilde{n}$ is small and the AK-SVD avoids it. However, since matrix multiplication is an operation with much parallelism, we can afford it on the GPU.

We introduce also the modification proposed in [14], to perform several cycles of updates for each sparse representation stage. The variable u from step 7 is 1 in the standard AK-SVD, but one or two more dictionary update cycles may be useful. Convergence speed may be gained if the parallel updates are faster than the OMP representations. In principle, the values of u and $\tilde{n}$ could change during the AK-SVD main iterations.

Algorithm 1: PAK-SVD

Data: initial dictionary $D \in \mathbb{R}^{p \times n}$
signals set $Y \in \mathbb{R}^{p \times m}$
target sparsity s
number of K-SVD iterations K
number of parallel atoms $\tilde{n}$
number of parallel signals $\tilde{m}$
number of update iterations u
Result: trained dictionary D
sparse representations $X \in \mathbb{R}^{n \times m}$

1 **for** $k \leftarrow 1$ **to** K **do**
2 $G = D^T D$, in parallel
3 $H = D^T Y$, in parallel
4 **for** $\ell \leftarrow 1$ **to** $m/\tilde{m}$ **do**
5 **for** $j \leftarrow (\ell - 1)\tilde{m} + 1$ **to** $\ell\tilde{m}$, *in parallel* **do**
6 $X_j = BOMP(G, H_j, s)$
7 **for** $i \leftarrow 1$ **to** u **do**
8 **for** $\ell \leftarrow 1$ **to** $n/\tilde{n}$ **do**
9 $E = Y - DX$
10 **for** $j \leftarrow (\ell - 1)\tilde{n} + 1$ **to** $\ell\tilde{n}$, *in parallel* **do**
11 $\mathcal{I}$ = {indices of signals whose representations use the j-th atom}
12 $F = E_{\mathcal{I}} + D_j X_{j,\mathcal{I}}$
13 $D_j = F X_{j,\mathcal{I}}^T / \|F X_{j,\mathcal{I}}^T\|_2$
14 $X_{j,\mathcal{I}} = F^T D_j$

2.3. OpenCL implementation details

The OpenCL platform allows us to execute small functions (kernels) in parallel on a chosen number of processing elements (PE), or work-items, within the compute units located on the OpenCL device [1]. These PEs are organized in an

n-dimensional space that can be set up in different ways for each kernel. The n-dimensional space is split into local work-groups, corresponding to compute units; PEs in a work-group can better share common resources. For example, in 2D, we can denote the n-dimensional range definition as NDR($\langle x_g, y_g \rangle$, $\langle x_l, y_l \rangle$). There are $x_g \times y_g$ PEs, organized on work-groups of size $x_l \times y_l$, running the same kernel.

Matrix multiplication. Steps 2, 3 and 9 from the PAK-SVD algorithm were implemented in OpenCL through a dedicated block-based full-matrix multiplication kernel.

The kernel semantics follow that of the classic GEMM BLAS operation. The input matrices as well as the result matrix are kept in global memory. The operations for calculating a block of the resulting matrix were performed within a work-group.

We mapped the elements of the resulting matrix in the shape of a 2-dimensional global space that is further split into block-sized work-groups. The optimal block size on our GPU device was found to be of 64×64 PEs, thus for a result matrix $A \in \mathbb{R}^{n \times m}$ we defined our n-dimensional space as: NDR($\langle n, m \rangle$, $\langle 64, 64 \rangle$).

Before doing the actual multiplication, each work-item within a work-group copies an element from the input block sub-matrices into local memory. This extra step is paid in full by faster memory access time during matrix multiplication. Further on, we use vectorized types for the arithmetic operations. On our device, float4 proved to be the fastest type.

Orthogonal Matching Pursuit. All the operations required for the sparse representation of a single signal, step 6 in PAK-SVD, were packed in and implemented by a single OpenCL kernel.

The input matrices as well as the resulting sparse signal are kept in global memory. The BLAS operations required for performing the Cholesky update and for recalculating the residual are done sequentially inside the BOMP kernel, not through a separate call to the BLAS kernel. Due to the rather small size of the matrices involved in these operations, measurements showed that using a dedicated kernel (as for steps 2, 3 and 9) does not even begin to pay for the required GPU IO. In-lining proved to be a lot faster.

The main obstacles we encountered while implementing step 6 were memory-bound. BOMP is a huge memory consumer and mostly due to auxiliary data. The necessary memory is of size $O(ns)$. Keeping all the auxiliary data in local memory would permit only the processing of one signal per compute-unit, corresponding to an $NDR(\langle \tilde{m} \rangle, \langle 1 \rangle)$ splitting. This would be wasteful as it would not reach full GPU occupancy and thus it would not cover the global memory latency costs.

After trying several work-group sizes, like 64, 128 and 256, we decided to leave the decision to the GPU scheduler, by using $NDR(\langle \tilde{m} \rangle, \langle \text{any} \rangle)$. This solution appears the best in our case. As mentioned before, we took $\tilde{m} = m$.

Atoms update. The atom updating process, steps 11–14 in PAK-SVD, was implemented through another dedicated OpenCL kernel. The input matrices E and X are kept in global memory as well as the existing dictionary D. However, the atom D_j is transferred to private memory, where it is kept during the update operations. This poses no problem due to the reasonable problem size (we used $p \leq 64$).

On the other hand, storing a list of indices $\mathcal{I}$ (step 11) turns out to be difficult, since the number of indices varies a lot, depending on how much an atom is used in sparse signal representations. The GPU we have used has 32k local memory, which allows storage of about 8000 indices. If the number of signals is smaller, we use local memory, which is the ideal solution. Otherwise, the set $\mathcal{I}$ is stored as a global variable and hence global memory access latency diminishes the performance of the update stage.

In both cases we define a 1-dimensional space of $\tilde{n}$ global PEs. We leave defining the work-group size to the GPU scheduler by using $NDR(\langle \tilde{n} \rangle, \langle \text{any} \rangle)$.

The BLAS operations required for performing steps 12–14 were all done inside the update kernel in a sequential fashion for the same reasons enumerated when describing step 6 (BOMP).

BLAS issues. For further optimization we tried using the BLAS library for OpenCL from AMD. While probably good for one-time use scenarios, it did not give good performance in our case, which needed multiple calls for mostly quickly changing, small sized, data. The loss in transfer times between host and OpenCL memory was not compensated at all by the parallel computations. We were hence obliged to implement our own versions of BLAS operations, described above, for both big and small data sets.

3. RESULTS AND PERFORMANCE

We used colored and gray scale bitmap images for the training signals, taken from the USC-SIPI [15] image database (e.g. barb, lena, boat, etc.). The images were normalized and split into random 8×8 blocks representing the patches. The initial dictionary was built similarly or by generating random atoms; when comparing different algorithms, the initialization was always the same.

As a rule, we chose the dimensions as powers of two because this way the data objects and the work-loads are easier divided and mapped across the NDRs without the need for padding. We picked $p = 64$ and $n \in \{64, 128, 256, 512\}$, while we ran through a wide range of signal set dimensions (1024–131072). While we did most of the profiling around $s = 8$, we consistently investigated $s \in \{4, 6, 8, 10, 12\}$ when it came to minimizing the error. For parallelism, we used almost all the time $\tilde{m} = m$ while we walked $\tilde{n}$ from 1, in increments of powers of 2, up to n.

We tested our OpenCL implementation of PAK-SVD on an ATI FirePro V8800 (FireGL V) card from AMD, running at a maximum clock frequency of 825MHz, having 1600

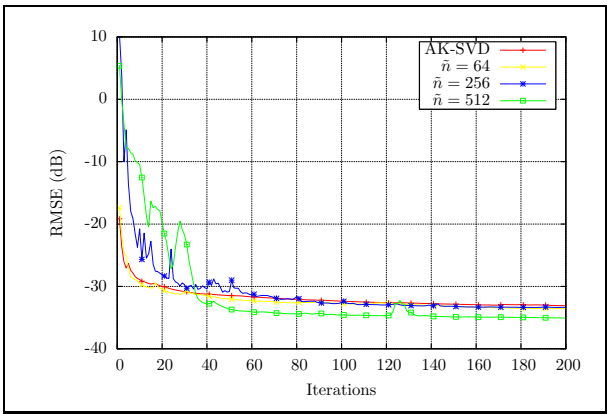

Fig. 1: Error evolution for $m = 16384$, $n = 512$, $s = 12$.

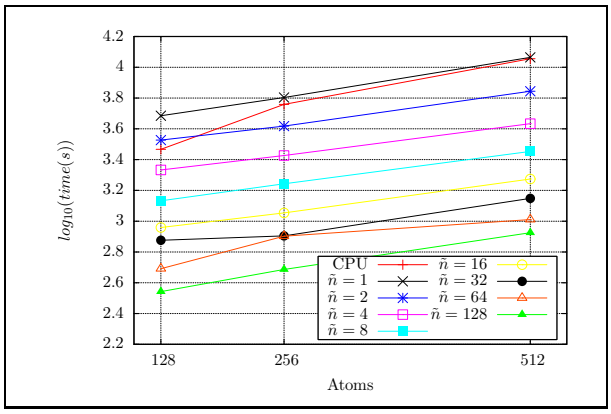

Fig. 2: Execution times for $m = 16384$, $s = 10$, $K = 200$.

Table 1. Final errors for AK-SVD and PAK-SVD with $\tilde{n} = n$.

		n = 128		256		512	
		AK	PAK	AK	PAK	AK	PAK
s	4	0.0425	0.0407	0.0385	0.0387	0.0376	0.0372
	6	0.0374	0.0349	0.0334	0.0316	0.0311	0.0297
	8	0.0345	0.0306	0.0294	0.0272	0.0259	0.0245
	10	0.0322	0.0276	0.0276	0.0239	0.0233	0.0206
	12	0.0319	0.0249	0.0254	0.0205	0.0221	0.0176

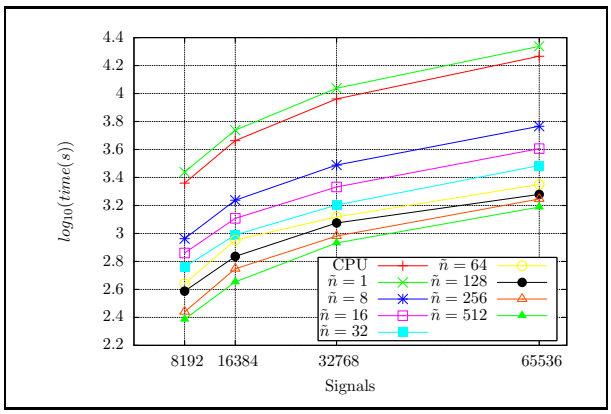

Fig. 3: Execution times for $n = 512$, $s = 8$, $K = 100$.

streaming processors, 2GB global memory and 32KB local memory. Also, the CPU tests for our AK-SVD C implementation were made on an Intel i7-3930K CPU running at a maximum clock frequency of 3.2GHz.

Figure 1 presents a typical example of error evolution during the iterative process, for AK-SVD (KSVD-Box implementation for Matlab by the first author of [12]) and PAK-SVD with various values of $\tilde{n}$. The RMSE is $\|E\|_F/\sqrt{mn}$, with E defined by (1). While for AK-SVD the error is typically decreasing, for PAK-SVD with large $\tilde{n}$ it occasionally increases, sometimes significantly. This behavior is milder for smaller sparsity level s. However, despite the erratic beginning, PAK-SVD tends to reach a smaller final error.

This is confirmed by the results from Table 1, where the error after $K = 200$ iterations is shown for PAK-SVD with $\tilde{n} = n$ and AK-SVD, for several values of the dictionary size and target sparsity. In all cases but one, the PAK-SVD with completely parallel atom updates outperforms AK-SVD. This is somewhat unexpected, but confirmed by other experiments, not reported here, on different subsets of images. Although Gauss-Seidel iterations are usually better than Jacobi, this is not always the case, as confirmed by our examples. However, more investigation is necessary for other types of data.

We now shift our focus towards the execution time of PAK-SVD, looking first on the influence of $\tilde{n}$. As expected, we can see in figure 2, where $m = 16384$, that larger $\tilde{n}$ gives better results, producing a speed-up of at least 10 when going from $\tilde{n} = 1$ to $\tilde{n} = n$.

A similar behavior is visible in figure 3, where we kept a fixed dictionary size of $n = 512$ atoms and increased the number of patches in the signal set. Naturally, the speed-up with respect to the case $\tilde{n} = 1$ grows as the size of the problem increases.

To put things in perspective, we implemented AK-SVD in C for comparing times spent on the CPU versus times spent on the GPU. We tried to keep the instructions between the two versions the same wherever possible (the update stage obviously had to vary). As it can be seen in figures 2 and 3, PAK-SVD is around 10 times faster when $\tilde{n} = n$. It outperforms the CPU version except for the case where $\tilde{n} = 1$, which is to be expected due to the error calculation and compensation for each atom update and also due to the low GPU utilization. For the largest problem, with $n = 512$, $m = 65536$, the speed-up is 11.98.

We also compared the performance of KSVD-Box to our OpenCL implementation of PAK-SVD. The results varied based on the hardware underneath, but on new desktops with multicore processor KSVD-Box has comparable or even

smaller execution times than PAK-SVD with $\tilde{n} = n$. This is not surprising, due to the heavily-optimized full and sparse matrix routines available in Matlab and its multi-threading capabilities. We expect that further development of GPU software for numerical computations will increase the performance of our implementation.

Our measurements showed that a single dictionary update (steps 8–14 of PAK-SVD) can be 2 to 3 times faster than the sparse representation stage. This observation lead to modifying the original K-SVD algorithm so that multiple update rounds can be performed during the same SVD iteration. So our experiments included varying the parameter u (step 7). Taking as a reference the case where $u = 1$, we found that bumping the number of rounds to $u = 2$ or $u = 3$ does not significantly increase the execution time. So incrementing u allows us to either reach the same approximation error faster or to obtain a better approximation in a similar time as $u = 1$. We obtained the best results by keeping $\tilde{n} \leq 64$ and by using $u = 1$ for the first 50 iterations, and only afterwards increasing the number of dictionary updates per SVD iteration ($u \geq 2$). We noticed that when $u \geq 2$ from the start with large dictionaries ($n = 512$) and high parallelism ($\tilde{n} \geq 128$), the error actually increases at the second or third consecutive update.

4. CONCLUSIONS AND FUTURE WORK

We have proposed a parallel version of the AK-SVD algorithm, with two kinds of improvements. First, the atoms updates are performed in parallel, covering the whole range from the sequential Gauss-Seidel-style iterations of the standard algorithm to fully parallel Jacobi-style iterations. Second, we have implemented our parallel algorithm (PAK-SVD) in OpenCL and tested its behavior on GPU.

Training dictionaries for image representation, we have noticed that PAK-SVD is able to produce smaller representation error than AK-SVD, although its convergence is more erratic in the beginning. For the sizes that we have tried, the OpenCL implementation is up to 12 times faster than its sequential counterpart.

Further work will be devoted to the refinement of the implementation, for example by investigating the use of sparse matrices. The current experience will be used for exploring parallel versions of other algorithms for training dictionaries for sparse representations.

REFERENCES

[1] Khronos OpenCL Working Group, *The OpenCL Specification, Version 1.2, Revision 19*, Khronos Group, 2012.

[2] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming," *Parallel Computing*, vol. 38, pp. 391–407, 2014.

[3] M. Martins, G. Falcao, and I.M. Figueiredo, "Fast Aberrant Crypt Foci Segmentation on the GPU," in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 2013, pp. 1113–1117.

[4] G. Condello, P. Pasteris, D. Pau, and M. Sami, "An OpenCL-based feature matcher," *Signal Proc. Image Comm.*, vol. 28, pp. 345–350, 2013.

[5] C. Garcia, G. Botella, F. Ayuso, M. Prieto, and F. Tirado, "Multi-GPU based on multicriteria optimization for motion estimation system," *EURASIP J. Adv. Signal Proc.*, vol. 23, 2013.

[6] K. Hwang and W. Sung, "Load Balanced Resampling for Real-Time Particle Filtering on Graphics Processing Units," *IEEE Trans. Signal Proc.*, vol. 61, no. 2, pp. 411–419, Jan. 2013.

[7] K. Konishi, "Parallel GPU Implementation of Null Space Based Alternating Optimization Algorithm for Large-Scale Matrix Rank Minimization," in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 2012, pp. 3698–3692.

[8] P. Nagesh, R. Gowda, and B. Li, "Fast GPU Implementation of Large Scale Dictionary and Sparse Representation Based Vision Problems," in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 2010, pp. 1570–1573.

[9] R. Rubinstein, A.M. Bruckstein, and M. Elad, "Dictionaries for Sparse Representations Modeling," *Proc. IEEE*, vol. 98, no. 6, pp. 1045–1057, June 2010.

[10] K. Engan, S.O. Aase, and J.H. Husoy, "Method of optimal directions for frame design," in *IEEE Int. Conf. Acoustics Speech Signal Proc.*, 1999, vol. 5, pp. 2443–2446.

[11] M. Aharon, M. Elad, and A. Bruckstein, "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation," *IEEE Trans. Signal Proc.*, vol. 54, no. 11, pp. 4311–4322, Nov. 2006.

[12] R. Rubinstein, M. Zibulevsky, and M. Elad, "Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit," Tech. Rep. CS-2008-08, Technion Univ., Haifa, Israel, 2008.

[13] M. Sadeghi, M. Babaie-Zadeh, and C. Jutten, "Learning Overcomplete Dictionaries Based on Atom-by-Atom Updating," *IEEE Trans. Signal Proc.*, vol. 62, no. 4, pp. 883–891, Feb. 2014.

[14] L.N. Smith and M. Elad, "Improving Dictionary Learning: Multiple Dictionary Updates and Coefficient Reuse," *IEEE Signal Proc. Letters*, vol. 20, no. 1, pp. 79–82, Jan. 2013.

[15] A.G. Weber, "The USC-SIPI Image Database," 1997.