GPU Accelerated Sparse Representation of Light Fields

Gabriel Baravdish[1], Ehsan Miandji[1] and Jonas Unger[1]
[1]*Department of Science and Technology, Linköping University, Bredgatan 33, Norrköping, Sweden*
{gabriel.baravdish, ehsan.miandji, jonas.unger}@liu.se

Keywords: Light field compression, GPGPU computation, sparse representation

Abstract: We present a method for GPU accelerated compression of light fields. The approach is by using a dictionary learning framework for compression of light field images. The large amount of data storage by capturing light fields is a challenge to compress and we seek to accelerate the encoding routine by GPGPU computations. We compress the data by projecting each data point onto a set of trained multi-dimensional dictionaries and seek the most sparse representation with the least error. This is done by a parallelization of the tensor-matrix product computed on the GPU. An optimized greedy algorithm to suit computations on the GPU is also presented. The encoding of the data is done segmentally in parallel for a faster computation speed while maintaining the quality. The results shows an order of magnitude faster encoding time compared to the results in the same research field. We conclude that there are further improvements to increase the speed, and thus it is not too far from an interactive compression speed.

1 INTRODUCTION

Light field imaging has been an active research topic for more than a decade. Several new techniques have been proposed focusing on light field capture (Liang et al., 2008; Babacan et al., 2012), super-resolution (Wanner and Goldluecke, 2013; Choudhury et al., 2017), depth estimation (Vaish et al., 2006; Williem and Park, 2016), refocusing (Ng, 2005), geometry estimation (Levoy, 2001), and display (Jones et al., 2016; Wetzstein et al., 2012). A light field represents a subset of the Plenoptic function (Adelson and Bergen, 1991), where we store the outgoing radiance at several spatial locations $(\mathbf{r}_i, \mathbf{t}_j)$, and along multiple directions $(\mathbf{u}_\alpha, \mathbf{v}_\beta)$, as well as as the spectral data λ_γ. Note that here we consider a discrete function $l(\mathbf{r}_i, \mathbf{t}_j, \mathbf{u}_\alpha, \mathbf{v}_\beta, \lambda_\gamma)$ containing the light field of a scene. The ongoing advances in sensor design, as well as computational power, have enabled imaging systems capable of capturing high resolution light fields along angular and spatial domains. A key challenge in such imaging systems is the extremely large amount of data produced. Difficulties arise in terms of bandwidth during the capturing phase and the storage phase. Fast and high quality compression is essential for existing imaging systems, as well as future designs due to the rapid increase in the amount of data produced.

In (Miandji et al., 2013) and (Miandji et al., 2015), a learning based method for compression of light fields and surface light fields is proposed. After dividing a collection of light fields into small two dimensional (2D) patches (i.e. matrices), a training algorithm computes a collection of orthogonal basis functions. These orthogonal basis functions are in essence code words that enable sparse representation (Elad, 2010) of light fields. We refer to these basis functions as *dictionaries*, a commonly used term in sparse representation literature (Aharon et al., 2006). The training process is performed once on a collection of light fields. Once the dictionaries are trained, the next step is to project the patches from a light field we would like to compress onto the dictionaries. The result is a set of sparse coefficients, which significantly reduces the storage cost. While the method produces a representation with a small storage cost and high reconstruction quality, the projection step is computationally expensive. This makes the utility of the algorithm for capturing light fields impractical.

In this paper we propose a GPU accelerated algorithm that enables the sparse representation of light field data sets for compression. This algorithm replaces the projection step discussed in (Miandji et al., 2013) and (Miandji et al., 2015), given a set of precomputed 2D dictionaries. Moreover, we show that our algorithm can be extended to higher dimensions, i.e. instead of using 2D patches, we use 5D patches for light fields. The higher dimensional method is shown to be favorable in terms of performance. While we focus on light fields, we believe our method can be used for variety of other large scale data sets in graphics, e.g. measured BTF and BRDF data sets.

2 TWO DIMENSIONAL COMPRESSION

Let $\{\mathbf{T}^{(i)}\}_{i=1}^{N} \in \mathbb{R}^{m_1 \times m_2}$ be a collection of patches extracted from a light field or light field video. The training method described in (Miandji et al., 2013) computes a set of K two dimensional dictionaries $\{\mathbf{U}^{(k)}, \mathbf{V}^{(k)}\}_{k=1}^{K}$, where $\mathbf{U}^{(k)} \in \mathbb{R}^{m_1 \times m_1}$ and $\mathbf{V}^{(k)} \in \mathbb{R}^{m_2 \times m_2}$. Using a constraint of sparsity during training, this model enables sparse representation of a light field patch in one dictionary, i.e. $\mathbf{T}^{(i)} = \mathbf{U}^{(k)}\mathbf{S}^{(i)}(\mathbf{V}^{(k)})^T$, where $\mathbf{S}^{(i)}$ is a sparse matrix.

We assume that a collection of dictionaries $\{\mathbf{U}^{(k)}, \mathbf{V}^{(k)}\}_{k=1}^{K}$ is trained using the method described in (Miandji et al., 2013) or (Miandji et al., 2015). With a slight abuse of notation, let $\{\mathbf{T}^{(i)}\}_{i=1}^{N} \in \mathbb{R}^{m_1 \times m_2}$ be a set of patches from a light field we would like to compress. Note that the training set and the data set we would like to compress are distinct. For compression, i.e. computing sparse coefficients for each patch, we proceed as follows: Each patch is projected onto all dictionaries as $\mathbf{S}^{(i,k)} = \mathbf{V}^{(k)}\mathbf{T}^{(i)}(\mathbf{U}^{(k)})^T$. Then we set a maximum of $m_1 m_2 - \tau$ elements of $\mathbf{S}^{(i,k)}$ with the smallest absolute value to zero, where τ is a user defined sparsity parameter. The nullification is also controlled with a threshold parameter on the representation error, denoted ε. The coefficient matrix among the set $\{\mathbf{S}^{(i,k)}\}_{k=1}^{K}$ that is the sparsest with least error is stored. Since each patch uses one dictionary among K dictionaries, we also store the index of the dictionary used for each patch, which is called a *membership* index.

We compute the product $\mathbf{S}^{(p,k)} = \mathbf{V}^{(k)}\mathbf{T}^{(p)}(\mathbf{U}^{(k)})^T$ for p patches in parallel on the GPU. This is done by launching equal amount of threads as the total number of elements among all patches. Then, we let each thread extract a row of a patch $\mathbf{T}^{(i)}$ and copy it to the on-chip cached memory, also called *shared memory*. Each thread computes the inner product between the row of the patch and the column of the given dictionary. The data is stored as one long vector on the memory, and to find the corresponding row of a patch i and a thread j, we use the expression in Equation 2.

3 MULTIDIMENSIONAL COMPRESSION

For multidimensional compression, we seek the most sparse representation of an n-dimensional patch $\boldsymbol{\mathcal{T}}^{(i)} \in \mathbb{R}^{m_1 \times m_2 \times \ldots \times m_n}$ under a given set of K multidimensional dictionaries $\left\{\mathbf{U}^{(1,k)}, \ldots, \mathbf{U}^{(n,k)}\right\}_{k=1}^{K}$. Algorithm 1 is an extension of the greedy method presented in (Miandji et al., 2015) that achieves this goal. The algorithm computes a dictionary membership index $a \in [1, \ldots, K]$, where K is the number of dictionaries, and sparse coefficients $\boldsymbol{\mathcal{S}}^{(i)}$ using a threshold for sparsity τ and a threshold for error ε. Note that Algorithm 1 is repeated for all the patches $\{\boldsymbol{\mathcal{T}}^{(i)}\}_{i=1}^{N}$. There are several steps in the algorithm that are computationally expensive and need to be implemented in such a way that utilize the highly parallel architecture of modern GPUs. In particular, step 3 of the algorithm perfoms multiple n-mode products between a tensor and a matrix. Similarly, in step 6 we have a similar operation, as well as an expensive norm computation. In what follows, we will present algebraic manipulations of the computationally expensive steps of the algorithm, as well as GPU-friendly implementation techniques.

Linear algebra computations are often memory bounded. Therefore, to minimize the memory transactions between the CPU and GPU we compute and store all $\hat{N} \leq N$ number of data points in parallel on the GPU, where $\hat{N}$ is the maximum number of data points that can fit in the GPU memory and N is the total number of data points. Moreover, we describe the procedure to parallelize and compute the tensor-matrix product on the GPU in section 3.1.

3.1 Computing the n-mode Product on the GPU

A tensor is an n-dimensional array of order n, or n modes. A *fiber* is specified by fixing every index but one of a tensor. We use the same tensor notation and colon notation as (Kolda and Bader, 2009), where $\boldsymbol{\mathcal{X}}(i_1, :, i_3)$ represents a fiber with all elements along 2-mode of a third-order tensor.

The definition of the tensor-matrix product, as with the more common matrix-vector product, is an inner product between every fiber in the tensor along the n-th dimension and every column in the corresponding matrix. Thus, the task is to extract each fiber of the tensor and perform an inner product.

In order to efficiently extract a fiber along the n-mode we unfold a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ to a matrix.

Algorithm 1 Compute coefficients and the membership index.

Require: A patch $\mathcal{T}_i$, error threshold ε, sparsity τ, and dictionaries $\left\{\mathbf{U}^{(1,k)},\ldots,\mathbf{U}^{(n,k)}\right\}_{k=1}^{K}$
Ensure: The membership index a and the coefficient tensor $\boldsymbol{\mathcal{S}}$

1: $\mathbf{e} \in \mathbb{R}^K \leftarrow \infty$ and $\mathbf{z} \in \mathbb{R}^K \leftarrow \mathbf{1}$
2: **for** $k = 1 \ldots K$ **do**
3: $\quad \boldsymbol{\mathcal{X}}^{(k)} \leftarrow \boldsymbol{\mathcal{T}}^{(k)} \times_1 \left(\mathbf{U}^{(1,k)}\right)^T \cdots \times_n \left(\mathbf{U}^{(n,k)}\right)^T$
4: $\quad$ **while** $\mathbf{z}_k \leq \tau$ and $\mathbf{e}_k > \varepsilon$ **do**
5: $\quad\quad \boldsymbol{\mathcal{Y}} \leftarrow$ Nullify $(\prod_{j=1}^{n} m_j) - \mathbf{z}_k$ smallest element of $\boldsymbol{\mathcal{X}}^{(k)}$
6: $\quad\quad \mathbf{e}_k \leftarrow \left\| \boldsymbol{\mathcal{T}}^{(i)} - \boldsymbol{\mathcal{Y}} \times_1 \mathbf{U}^{(1,k)} \cdots \times_n \mathbf{U}^{(n,k)} \right\|_F^2$
7: $\quad\quad \mathbf{z}_k = \mathbf{z}_k + 1$
8: $\quad$ **end while**
9: $\quad \boldsymbol{\mathcal{X}}^{(k)} \leftarrow \boldsymbol{\mathcal{Y}}$
10: **end for**
11: $a \leftarrow$ index of $\min(\mathbf{z})$
12: **if** $\mathbf{z}_a = \tau$ **then**
13: $\quad a \leftarrow$ index of $\min(\mathbf{e})$
14: **end if**
15: $\boldsymbol{\mathcal{S}}^{(i)} \leftarrow \boldsymbol{\mathcal{X}}^{(a)}$

Let

$$I = \prod_{n=1} I_n \quad \text{and} \quad \hat{I}_n = I/I_n, \quad n \in \{1,\ldots,N\} \tag{1}$$

represent the total amount of elements in a tensor $\boldsymbol{\mathcal{X}}$. We unfold $\boldsymbol{\mathcal{X}}$ to a $I \times \hat{I}_n$ matrix and map each element with the subscripts $(i_1, i_2, \ldots, i_n)$ to the matrix index (i_n, j), see Equation 2. By unfolding the tensor we ensure that the n-th order fibers are rearranged to be the columns of the resulting matrix. By the same approach the matrix can be unfolded to a single column vector. This vectorization allows us to take advantage of the linear memory structure and efficient memory accesses on the GPU. Memory linearization is important for minimal cost of memory transaction from and to the global memory.
A tensor element $x_{i_1,i_2,\ldots,i_N}$ is mapped to the entry (i_n, j) of $\mathbf{X}_{(\mathrm{n})}$ by

$$j = \sum_{\substack{k=0 \\ k \neq n}}^{N} i_k J_k, \quad \text{where} \quad J_k = \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m. \tag{2}$$

For tensor sizes that are small enough, it is more convenient to let each thread compute the inner product of a fiber and all the columns in a dictionary matrix. This is due to the computational overhead of thread collaborations. When traversing along the n-th dimension, the group of threads in a block stores m_n tensor fiber elements and m_n^2 dictionary elements from the global memory to the shared memory, see Figure 1. The inner product between a fiber and all the columns in a dictionary is then computed on the shared memory. The resulting fiber is stored at the same location on the global memory due to the dictionary being squared matrices.

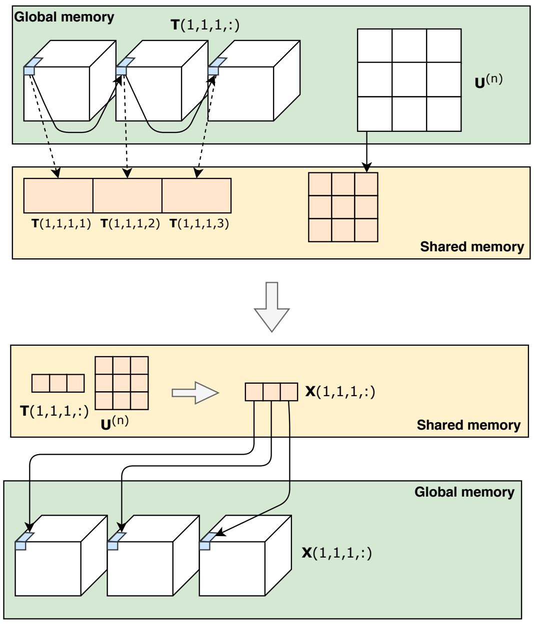

Figure 1: The n-mode product on the GPU. Here $\boldsymbol{\mathcal{T}}$ is a 4D-tensor and the first thread is traversing along the fourth dimension. Observe that this is just an illustration of the concept, the data points are not explicitly stored as tensors on the global memory.

3.2 Computing Sparse Coefficient Tensors

In this section we will cover how to transform the norm of the n-mode product of a sparse tensor into almost a single instruction, see line 10 in Algorithm 1.

The Frobenius norm of a tensor $\boldsymbol{\mathcal{X}}$ is the square root of the sum of the squares of all its elements

$$\|\boldsymbol{\mathcal{X}}\|_F = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} x_{i_1 i_2 \ldots i_N}^2}. \tag{3}$$

Let the norm of a tensor be defined as Equation 3. Let

$$\begin{aligned} \boldsymbol{\mathcal{T}} &= \boldsymbol{\mathcal{X}} \times_1 \mathbf{U}^{(1,k)} \cdots \times_n \mathbf{U}^{(n,k)} \\ \boldsymbol{\mathcal{X}} &= \boldsymbol{\mathcal{T}} \times_1 (\mathbf{U}^{(1,k)})^T \cdots \times_n (\mathbf{U}^{(n,k)})^T \end{aligned} \tag{4}$$

and

$$\begin{aligned}\tilde{\mathcal{T}} &= \mathcal{Y} \times_1 \mathbf{U}^{(1,k)} \cdots \times_n \mathbf{U}^{(n,k)} \\ \mathcal{V} &= \mathcal{X} - \mathcal{Y} \\ <\mathcal{Y}, \mathcal{V}> &= 0\end{aligned} \tag{5}$$

, such that $\mathcal{Y}$ is a sparse version of $\mathcal{X}$, $\left\{\mathbf{U}^{(1,k)} \ldots \mathbf{U}^{(n,k)}\right\}_{n=1}^{N}$ form orthonormal basis and $\mathcal{V}$ is the complementary dense version of $\mathcal{X}$ with respect to $\mathcal{Y}$.

We will use the orthogonal invariance property from
$\left\{\mathbf{U}^{(1,k)} \ldots \mathbf{U}^{(n,k)}\right\}_{n=1}^{N}$, which preserves the norm, i.e

$$\begin{aligned}\|\mathcal{X}\|^2 &= <\mathcal{X}, \mathcal{X}> \\ &= <\mathcal{X}, \mathcal{T} \times_1 (\mathbf{U}^{(1,k)})^T \cdots \times_n (\mathbf{U}^{(n,k)})^T> \\ &= <\mathcal{X} \times_1 \mathbf{U}^{(1,k)} \cdots \times_n \mathbf{U}^{(n,k)}, \mathcal{T}> \\ &= <\mathcal{T}, \mathcal{T}> \\ &= \|\mathcal{T}\|^2\end{aligned} \tag{6}$$

By using Equation 4 and Equation 5 we can now define the problem as

$$\|\mathcal{T} - \tilde{\mathcal{T}}\|^2 = \|\mathcal{T}\|^2 - 2 <\mathcal{T}, \tilde{\mathcal{T}}> + \|\tilde{\mathcal{T}}\|^2, \tag{7}$$

and for the inner product we have that

$$\begin{aligned}<\mathcal{T}, \tilde{\mathcal{T}}> &= <\mathcal{T}, \mathcal{Y} \times_1 \mathbf{U}^{(n,k)} \cdots \times_n \mathbf{U}^{(1,k)}> \\ &= <\mathcal{T} \times_1 (\mathbf{U}^{(1,k)})^T \cdots \times_n (\mathbf{U}^{(n,k)})^T, \mathcal{Y}> \\ &= <\mathcal{X}, \mathcal{Y}> \\ &= <\mathcal{V} + \mathcal{Y}, \mathcal{Y}> \\ &= <\mathcal{V}, \mathcal{Y}> + <\mathcal{Y}, \mathcal{Y}> \\ &= 0 + <\mathcal{Y}, \mathcal{Y}> \\ &= \|\mathcal{Y}\|^2.\end{aligned} \tag{8}$$

With the property of orthogonality in Equation 6 and Equation 8 we can rewrite Equation 7 to

$$\begin{aligned}\|\mathcal{T} - \tilde{\mathcal{T}}\|^2 &= \|\mathcal{T}\|^2 - 2\|\mathcal{Y}\|^2 + \|\mathcal{Y}\|^2 \\ &= \|\mathcal{T}\|^2 - \|\mathcal{Y}\|^2\end{aligned} \tag{9}$$

We can now take advantage of the result in Equation 9 in our iterative routine. Since $\mathcal{Y}$ is sparse and we iteratively include one element at a time from $\mathcal{X}$, we can break this down to an element-wise update. We have from Equation 3, together with a linear index j of P as the total number of elements, that

$$\begin{aligned}\|\mathcal{T}\|_F^2 &= \sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} t_{i_1 i_2 \ldots i_N}^2 \\ &= \sum_{j=1}^{P} t_j^2.\end{aligned} \tag{10}$$

By putting Equation 9 and Equation 10 together we have

$$\|\mathcal{T}\|_F^2 - \|\mathcal{Y}\|_F^2 = \sum_{j=1}^{P} t_j^2 - \sum_{l=1}^{P} y_j^2. \tag{11}$$

We exploit the sparse structure of $\mathcal{Y}$ and iteratively add one element at a time from $\mathcal{X}$. Let $0 < M < \tau$, where M is the number of nonzero coefficients in the sparse tensor $\mathcal{Y}$ at a specific iteration and τ is the sparsity paremeter. Then, from Equation 9 and the formulation of Equation 11 together with the index l as the location to the nonzero coefficients, we finally have

$$\|\mathcal{T}\|_F^2 - \|\mathcal{Y}\|_F^2 = \sum_{j=1}^{N} t_j^2 - \sum_{l=1}^{M} y_l^2. \tag{12}$$

4 RESULTS

The results for the GPU implementation were achieved with Nvidia GeForce GTX Titan Xp and Intel Xeon CPU W3670 at 3.2 GHz. Since the computations were performed on the GPU only one CPU core was used.

The timings for the CPU version were obtained by a machine with four Xeon E7-4870, a total of 40 cores at 2.4 GHz. The data sets we used to evalute our method were acquired by Stanford University (Computer Graphics Laboratory, 2018). For the training of the dictionaries we used the following light fields: *Lego Truck*, *Chess*, *Eucalyptus Flowers*, *Jelly Beans*, *Amethyst*, *Bunny*, *Treasure Chest* and *Lego Bulldozer*. In order to create 5D data points we used the central 8x8 views of the light fields. The data points of each data set have the dimensions m_1 = 5, m_2 = 5, m_3 = 3, m_4 = 8 and m_5 = 8.
For the testing set we used *Lego Knights* (1024x1024 image resolution), *Tarot Cards and Crystal Ball* (1024x1024 image resolution) and *Bracelet* (1024x680 image resolution), see Figure 2. We used sparsity parameter τ = 300, τ = 390, τ = 412, respectively for the three data sets. We set the threshold parameter $\varepsilon = 5 \times 10^{-5}$, $\varepsilon = 5 \times 10^{-5}$ and $\varepsilon = 7 \times 10^{-5}$, respectively.

We show the encoding time of the presented method, see Table **??**, where we compare CPU time and GPU time. We evaluated our method with $K = 64$ number of dictionaries. The total encoding time for the CPU for the three data sets are 124 seconds, 122 seconds and 83 seconds, respectively. Comparing this to our GPU implementation we get 8.5 seconds, 8.3 seconds and 5.2 seconds. Normalizing these results

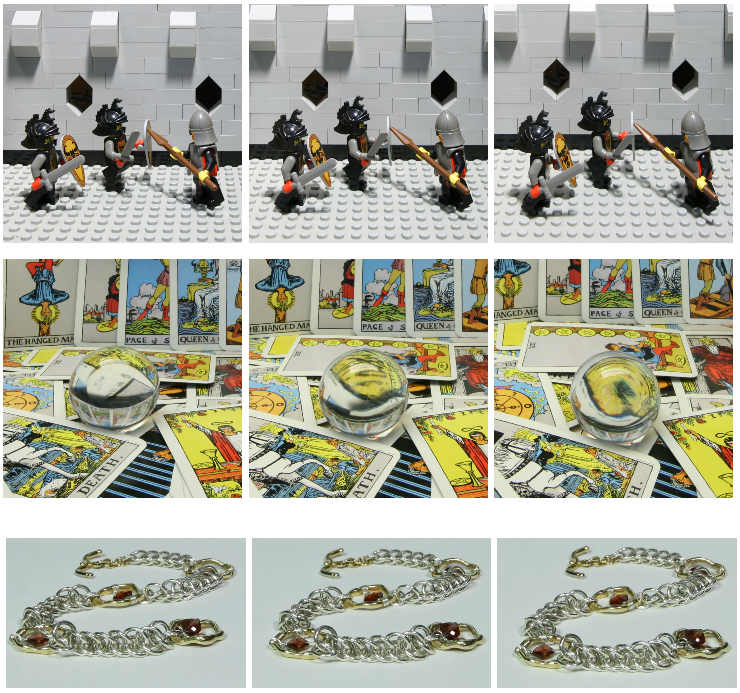

Figure 2: Reference views of the *Lego Knights*, *Tarot Cards and Crystal Ball* and *Bracelet* data sets.

Data set	**GPU Time** (s)	**CPU Time** (s)	**Speedup**
Lego Knights	8.5	124	x14.54
Tarot Cards	8.3	122	x14.70
Bracelet	5.2	83	x15.90

to $K = 1$, we get the following timings per dictionary: 133.2 ms for *Lego Knights*, 129.8 ms for *Tarot Cards and Crystal Ball* and 81.6 ms for *Bracelet*. Compared to the CPU timings: 1937 ms, 1906 ms and 1296 ms, respectively. As seen in Table **??**, we get a significant computation speedup by processing the data points segmentally in parallel on the GPU.

5 CONCLUSIONS

We have presented a GPU accelerated light field compression method. The implemented method scales in both memory and speed for higher dimensions - which leads to faster computations, not only by faster GPUs, but also by GPUs with larger memory. With an order of magnitude faster computation speed, we are able to produce a high quality compression that is equivalent to the results of similar work in the research field.

We also show that computations on a single GPU outperforms even massively parallel CPUs. For even faster performance, multiple GPUs can be used simultaneously.

With very few changes we can use same implementation for the training phase on the GPU. This would accelerate the creation of the dictionaries more.

REFERENCES

Adelson, E. H. and Bergen, J. R. (1991). The plenoptic function and the elements of early vision. In *Computational Models of Visual Processing*, pages 3–20. MIT Press.

Aharon, M., Elad, M., and Bruckstein, A. (2006). k -svd: An algorithm for designing overcomplete dictionaries for sparse representation. *Signal Processing, IEEE Transactions on*, 54(11):4311–4322.

Babacan, S., Ansorge, R., Luessi, M., Mataran, P. R., Molina, R., and Katsaggelos, A. K. (2012). Compressive light field sensing. *IEEE Trans. on Image Processing*, 21(12):4746–4757.

Choudhury, B., Swanson, R., Heide, F., Wetzstein, G., and Heidrich, W. (2017). Consensus convolutional sparse coding. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 4290–4298.

Computer Graphics Laboratory, S. U. (2018). Stanford university - the (new) stanford light field archive. `http://lightfield.stanford.edu/`. Accessed: 2018-07-23.

Elad, M. (2010). *Sparse and Redundant Representations: From Theory to Applications in Signal and Image Processing*. Springer Publishing Company, Incorporated, 1st edition.

Jones, A., Nagano, K., Busch, J., Yu, X., Peng, H. Y., Barreto, J., Alexander, O., Bolas, M., Debevec, P., and Unger, J. (2016). Time-offset conversations on a life-sized automultiscopic projector array. In *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 927–935.

Kolda, T. G. and Bader, B. W. (2009). Tensor decompositions and applications. *SIAM Review*, 51(3):455–500.

Levoy, M. A. (2001). The digital michelangelo project. *Computer Graphics Forum*, 18(3):xiii–xvi.

Liang, C.-K., Lin, T.-H., Wong, B.-Y., Liu, C., and Chen, H. H. (2008). Programmable aperture photography: multiplexed light field acquisition. In *Proc. of ACM SIGGRAPH*, volume 27, pages 1–10.

Miandji, E., Kronander, J., and Unger, J. (2013). Learning based compression of surface light fields for real-time rendering of global illumination scenes. In *SIGGRAPH Asia 2013 Technical Briefs*, SA '13, pages 24:1–24:4, New York, NY, USA. ACM.

Miandji, E., Kronander, J., and Unger, J. (2015). Compressive image reconstruction in reduced union of subspaces. *Comput. Graph. Forum*, 34(2):33–44.

Ng, R. (2005). Light field photography with a hand-held plenoptic camera. *Computer Science Technical Report CSTR 2*, 11:1–11.

Vaish, V., Levoy, M., Szeliski, R., Zitnick, C. L., and Kang, S. B. (2006). Reconstructing occluded surfaces using synthetic apertures: Stereo, focus and robust measures. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2331–2338.

Wanner, S. and Goldluecke, B. (2013). Variational light field analysis for disparity estimation and super-resolution. *IEEE Transactions of Pattern analysis and machine intelligence*, 36(3).

Wetzstein, G., Lanman, D., Hirsch, M., and Raskar, R. (2012). Tensor displays: Compressive light field synthesis using multilayer displays with directional backlighting. *ACM Trans. Graph.*, 31(4):80:1–80:11.

Williem, W. and Park, I. K. (2016). Robust light field depth estimation for noisy scene with occlusion. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4396–4404.