J Real-Time Image Proc manuscript No.
(will be inserted by the editor)

Vasileios Avramelos[1] · Ruben Verhack[1,3] · Ignace Saenen[1] · Glenn Van Wallendael[1] · Bart Goossens[2] · Peter Lambert[1]

Highly Parallel Steered Mixture-of-Experts Rendering at Pixel-Level for Image and Light Field Data

Received: date / Revised: date

Abstract A novel image approximation framework called Steered Mixture-of-Experts (SMoE) was recently presented. SMoE has multiple applications in coding, scale-conversion, and general processing of image modalities. In particular, it has strong potential for coding and streaming higher dimensional image modalities that are necessary to leverage full translational and rotational freedom (6 Degrees-of-Freedom) in virtual reality for camera captured images. In this paper, we analyze the rendering performance of SMoE for 2D images and 4D light fields. Two different GPU implementations that parallelize the SMoE regression step at pixel-level are presented, including experimental evaluations based on rendering performance and quality. In this paper it is shown that on appropriate hardware, the OpenCL implementation can achieve 85fps and 22fps for respectively 1080p and 4K renderings of large models with more than 100.000 of Gaussian kernels.

Keywords Steered Mixture-of-Experts, image compression, light field rendering, real-time rendering, GPU acceleration

The research activities described in this paper were funded by IDLab (Ghent University - imec), Flanders Innovation & Entrepreneurship (VLAIO), the Fund for Scientific Research Flanders (FWO Flanders), and the European Union.

[1] Ghent University - imec - IDLab, Department of Electronic and Information Systems
Technologiepark - Zwijnaarde 19, 9052 Ghent, Belgium
Tel.: +3293314957
E-mail: vasileios.avramelos@ugent.be

[2] Ghent University - imec - IPI, Department of Telecommunications and Information Processing
Sint-Pietersnieuwstraat 41B, 9000 Ghent, Belgium

[3] Technische Universität Berlin, Communications Systems Group
Einsteinufer 17, 10587 Berlin, Germany

1 Introduction

The consumption of virtual reality (VR) for camera captured content (e.g. 360° video) is lagging behind on the use of VR experiences of computer generated scenes (e.g. in computer games and edutainment software). 360° video allows only rotational head movements around three perpendicular axes for the viewer, but disregards any translational movements in the same 3D coordinate space. To attain the sense of freedom of computer generated VR content, Six Degrees-of-Freedom (6DoF) are required, i.e. three translational movements (walking around and small sideways head movements) combined with three rotational movements (head rotations and tilts). Perceived as a virtual reality by humans when combined, the rendered 2D images are processed versions of the higher-dimensional light data that surrounds us. In terms of signal processing, we are presented with a high-dimensional sampling problem with nonuniform and nonlinear sample spacing and high-dimensional spatio-directionally varying sampling kernels [1]. The high-dimensional space is defined by the 5D plenoptic function [2]. However, when there are no occlusions (i.e. "open space" assumption), the 5D space can be reduced to the 4D light field [3, 4]. This assumption does not hold for 6DoF in large scenes, however, at the moment this is a widely used simplification [4].

One promising novel methodology that aims to provide full 6DoF has been introduced, namely *Steered Mixture-of-Experts* (SMoE). It directly models the underlying plenoptic function in a continuous, analytical form, or a lower-dimensional projections of this function [2]. It does so by identifying coherent regions in the coordinate space of the samples and optimizes local linear regressors for that segment in the coordinate space. The total regression corresponds to a smoothed piecewise linear approximation of the plenoptic function (or of a lower-dimensional projection). Currently, SMoE has been successfully applied for images, video, and 4D light fields, with competitive rate-distortion results for low- to mid-level bitrates [5–7]. The local regressors currently reported are only linear and thus modeling

very high spatial frequencies is challenging, however, the theory does not limit the nature of the local regressors and further developments are an active area of research.

Nevertheless, SMoE has some very beneficial properties for the distribution of 6DoF visual content compared to traditional image coding methods. For rendering it has three important properties. Firstly, view-rendering is very lightweight and pixels are coded independent from one another. Secondly, SMoE is a space-continuous representation, thus rendering at arbitrary resolution consists of merely sampling this function. Finally, all local light information in a certain point in the physical space is also localized in the SMoE model.

Digital image and video compression techniques have been an important field of research since the 1950s. Standardized image and video coders typically rely on a transform step (e.g. wavelet or DCT) and *Differential Pulse-Code Modulation* (DPCM) (e.g. intra-prediction, and motion compensation). As a result, the current state-of-the-art coders like *High Efficiency Video Coding (HEVC)* are based on hybrid transform/DPCM coding schemes which consist mainly of the above mentioned techniques [8]. The serial nature of these old paradigms (e.g., intra-prediction) makes it impossible to really achieve pixel-level parallelism. Fine grained parallelism is becoming more and more desirable in algorithms as modern hardware tends to increase the number of execution threads rather than the speed of those threads. Furthermore, traditional coding schemes based on dense sample/coefficient grids do not scale easily towards higher dimensional image modalities. Each dimension that is added (e.g. time dimension in video, or two angular dimensions in 4D light fields) lets the amount of samples to be stored grow exponentially with the dimensionality of the image modality.

The *Moving Picture Experts Group* (MPEG) has started efforts to standardize a 6DoF video format by the year 2021 [9]. They aim at a process with two phases: (1) identifying the most important 2D views, and (2) rely on view synthesis methods to render other 2D views at decoder side. The identified views are expected to be coded using the same hybrid DPCM/transform coding approaches [10]. We claim that there are two main concerns. First, the view synthesis may require considerable computational complexity at the decoder side. Secondly, the serial nature of the paradigms is far from optimal as the prediction order is much less evident. In video coding, frames can be buffered if a logical order exists between them, e.g. when the frames are time-consecutive. However, considering the freedom to select a particular point of view in a 6DoF VR experience, no such logical order exists. As such, buffering and differential coding become more challenging.

Based on the above observations, we present an early analysis of efficient pixel-level parallel decoders of the SMoE framework that were built in-house on *Graphics Processing Units* (GPUs) for this study. GPU programming is challenging and it requires considerable programming expertise. We are therefore interested in the performance of a highly tuned low-level OpenCL [11] implementation and a more high-level automatic scheduling implementation using *Quasar*. Quasar is a new, hardware-independent programming framework for programming heterogeneous computation devices [12, 13]. A key goal of Quasar is to help programmers focus on the design and optimization of hardware accelerated algorithms by solving hardware-specific implementation issues automatically. Two implementations through Quasar are presented here, as well as a dedicated OpenCL implementation, and a CPU C++ version. These versions are compared in terms of speed and numerical precision. A slower but numerically more precise implementation in MATLAB [14] was used as a ground truth to evaluate numerical precision. The evaluation is performed on SMoE models of 2D images and short-baseline 4D light fields (i.e. very limited translational freedom).

This article is organized as follows. In Sec. 2, we discuss related work. Next, in Sec. 3, a theoretical overview of the SMoE framework is presented, with a focus on the rendering phase. Sec. 4 details a general high-level architecture of a pixel-level parallelized SMoE renderer. In Sec. 5, an in-depth analysis is done considering the rendering speed versus the numerical stability of the renderer. Sec. 6 describes the various parallel reconstruction implementations developed for this work. Sec. 7 presents an overview of the experimental results when comparing the fast SMoE decoder in different parallel computing platforms. Finally, in Sec. 8 we present our conclusions and future work.

2 Related Work

The SMoE framework works on the same data-driven principles as other machine learning methods like *Support Vector Regression* (SVR), *Radial Basis Function* (RBF) networks, and *Artificial Neural Networks* (ANNs) [15, 16]. Recent advances in this domain, combined with the development of strong dedicated hardware (e.g., GPUs) have significantly increased the feasibility to create computationally efficient implementations of otherwise challenging algorithms [17].

The calculations of the loglikelihood in order to obtain the kernel weights $w_j(x)$ (See Eq. 8 in the following section) are also present in the Expectation step (E-step) of the Expectation-Maximization (EM) algorithm [18]. In the past, research has been done in order to accelerate the EM algorithm using GPUs [19]. However, these approaches were investigated in order to accelerate the segmentation of large amounts of data. Our goal on the other hand, is to perform regression of which calculating the weights $w_j(x)$ is only a part of.

In this work, we will evaluate the performance of pixel-level parallel rendering of light fields. Light field rendering has been around for decades but has recently been making a comeback [3, 4]. The main difference between *classical*

light field rendering [3] and our work, is that in classical light field rendering each pixel in a rendered virtual viewpoint is a linear combination of pixels in all captured views. This means that in order to render views, all captured views need to be in memory. This has always been a large drawback for light field rendering as it limits the scale of which it can deployed in. The light field models in SMoE can be seen as an intermediate processing step. In SMoE, we identify the underlying function that could give rise to these views. This function is built as a continuous statistical model of the captured views. The intended goal is to reduce the memory requirements by storing only the parameters of the model in memory instead. For static light fields, a single Gaussian (defined by its prior, center and covariance matrix) can represent thousands of pixels [20].

Despite the serial nature of video coding standards such as HEVC, parallelism in decoding/rendering is still pursued. HEVC relies on smart implementations, e.g. using a wavefront approach [8]. This ensures that blocks are decoded as soon as their dependencies are available. However, using 64-by-64 CTU blocks in a 1080p video only allows for only 15 decoding blocks. In the case of 32-by-32 CTU blocks, one can achieve 30 parallel streams. Such a scheme does fit multi-threading architectures, but is less suited for massively parallel architectures.

3 Steered Mixture-of-Experts

3.1 Introduction

Steered Mixture-of-Experts (SMoE) is a novel framework for approximation of image modalities with many applications, such as image modality coding, scale conversion (e.g. frame interpolation), and image description (e.g. depth estimation). An in-depth overview about SMoE for images and static 4D light fields is presented in [20]. Due to the sparse structure in SMoE, it is readily extensible towards higher dimensional image modalities, such as 6DoF content. This is in stark contrast to traditional image coding schemes which rely on dense sample-grid structures. Moreover, SMoE departs significantly from the conventional coding methods by operating in the spatial domain and thus not using any kind of transform coding. Instead of storing exactly the samples or the transform coefficients that define the image, this method relies on modeling the underlying generative function that could have given rise to the samples. Generally, this underlying function corresponds to lower-dimensional projections of the plenoptic function [2].

The function approximation of the underlying generative function is done by identifying coherent, stationary regions in the image modality. Each segment is modeled using a single N-dimensional entity, which we call a *kernel* or *component*. SMoE is based on the divide-and-conquer principle that is present in all *Mixture-of-Experts* (MoE) approaches. These methods are well-known in machine learning [21].

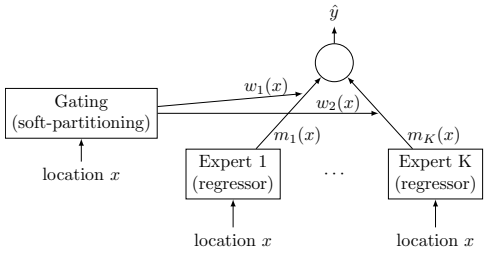


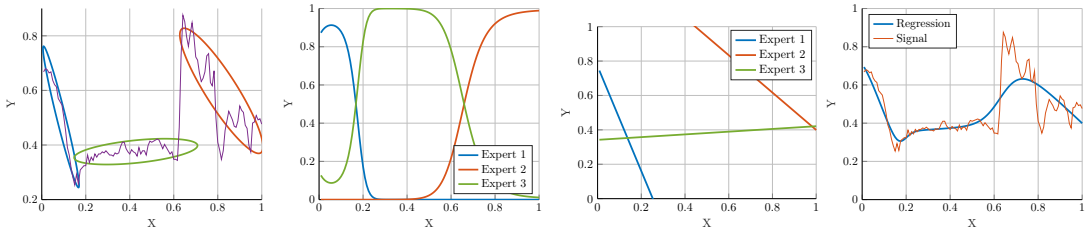
Fig. 1 Illustration of a classical Mixture-of-Experts with one layer for regression. The gating function soft-partitions the input space in regions where particular experts (in this case regressors) are the most influential.

The input space (in SMoE this is the *coordinate space*) is divided in soft-segments using a gating function. Local regressors (or *experts*) are sought that locally approximate the function optimally. The gating function then lets experts collaborate in segments where they are trustworthy. The gating network is illustrated in Fig. 1.

SMoE is based on the Bayesian, or "alternative" definition of the Mixture-of-Experts model [21]. The Bayesian Mixture-of-Experts approach jointly models the joint probability of the input space X and the output space Y using a *Gaussian Mixture Model* (GMM). Each Gaussian kernel then simultaneously defines the gating function (soft-segmentation of X) and the local regressors (through the conditional probability function $Y|X$).

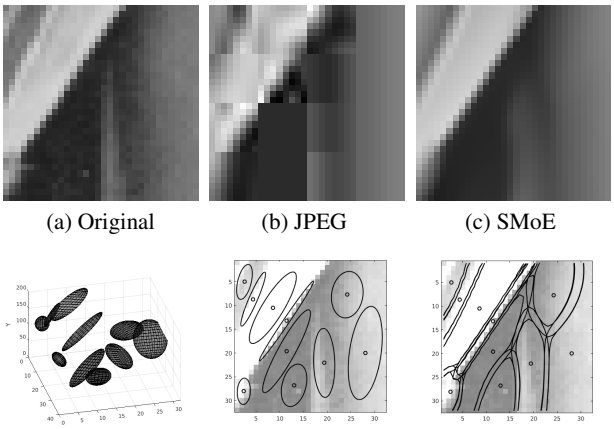
In SMoE, where the input space is the *coordinate space* (i.e. sample locations) and the output space is the *color space* (i.e. sample amplitudes), one such Gaussian then corresponds to one kernel as mentioned above. The gating function is thus defined by the probability of a coordinate to belong to a Gaussian, and each Gaussian simultaneously defines an expert function, namely the conditional color amplitudes, given a coordinate. In general, the SMoE allows to query the model at any sub-pixel coordinate to yield the most optimal amplitude in a Bayesian interpretation.

SMoE thus arrives at a sparse representation. The whole image modality is represented as a set of Gaussian kernels. These kernels are defined by their centers and their steering parameters. The coordinate space is 2D, 3D, or 4D in the case of respectively images, video, and static light fields [5–7]. The color space for color images is conventionally represented as a 3D space, e.g. RGB or YCbCr. As the Gaussians model the joint probability of the coordinate and color space, we thus arrive at 5D, 6D, and 7D Gaussian kernels. The parameters of these kernels are typically estimated using computational efficient variations of the *Expectation-Maximization* (EM) algorithm [18]. Due to this likelihood optimization, kernels will steer along the dimensions of the highest correlation, e.g., along spatial or temporal consistencies. Promising MSE-based modeling approaches to find the kernel parameters have been introduced very recently [24, 25]. In this paper, however, we work on likelihood optimized SMoE models without loss of generality.



(a) Signal and 2D GMM (b) Gating/Weights (c) Linear regressors (d) Signal and regression

Fig. 2 An illustrative 1D example of SMoE on a part of a scanline taken from *Lena* using three Gaussian kernels ($K = 3$). Both coordinate space and color space are 1D, thus resulting in a 2D joint probability density function (2a). Each Gaussian defines a linear regressor (2c). These regressors are summed using the weighting in (2b), as such we arrive at a smoothed piecewise linear reconstruction (2d).



(a) Original (b) JPEG (c) SMoE

(d) 3D GMM (e) Topview (f) Softmax

Fig. 3 An example of the modeling with 10 components and reconstruction of a 32x32 pixel crop from *Lena* (3a). For a grayscale image, the coordinate space X is 2D and the colorspace Y is 1D. Modeling the joint probability function of both X and Y using a Gaussian Mixture Model results in 3D Gaussian kernels (3d). Each kernel thus defines a 2D gradient as the *expert* function ($X \mapsto Y$). The gating function is defined by the soft-segmentation (3f). Both JPEG (3b) and SMoE (3c) are coded at 0.35 bpp [5].

Fig. 4 JPEG (left) vs SMoE (right) - reconstruction at 0.15 bpp. At this low bitrate JPEG yields an image with heavy block artefacts. SMoE typically results in an image which is excessively blurred and with fine details missing.

(a) Original (b) SMoE

Fig. 5 *Bikes* [22, 23] light field example (K=8960), showing a central view with $(a_1, a_2) = (7, 7)$. The original light field has $15x15x626x434$ samples. Consequently, each Gaussian kernel "covers" 6822 samples on average. (Mean $\text{PSNR}_{\text{YCbCr}}$: 30.71 dB, mean SSIM_{Y}: 0.86, evaluation as in [22]).

For illustration purposes, Fig. 2 depicts a SMoE network for the regression of samples from a 1D image scan line. Notice that for the 1D regression case we estimate 2D probability density functions (pdfs) using steered Gaussians. Fig. 2 depicts the samples and the resulting ellipsoids of each kernel and the three resulting linear hyper-planes of each model derived from the conditional distributions. The expert functions are linear functions in the 1D case which are gradients operating around the center of the expert component. Also shown are the associated three 1D window gating functions and the resulting *smoothed piecewise linear* regression function. The product of each window function with its respective linear regressor defines the final contribution of this expert to the regression function. The sum of all the gated linear regressors results in the final non-linear regression function [20].

Fig. 3 shows an example of the compression capability of the SMoE approach for coding a 32x32 pixel crop of Lena at 0.35 bits/sample in comparison to JPEG at same rate. Clearly, the edges are reconstructed with convincing quality and sharpness, using merely 10 components. In general, the framework achieves good performance for low-to-mid bitrates compared to the state-of-the-art, which is considerable taking into account the high difference in maturity (see Fig. 4). Fig. 5 illustrates a SMoE light field reconstruction using only 8960 kernels.

Fig. 6 illustrates the high-level coding process. The encoding step thus relies on an iterative optimization process using EM. Due to the specific structure of the data in image modalities, many heuristics can be used to arrive at an

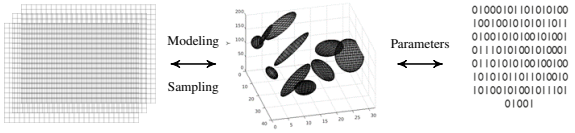


Fig. 6 A high-level view of the encoding scheme based on SMoE. Acquired sample grids are being modeled into a set of Gaussian kernels. In order to store this model only the parameters of these Gaussians need to be binarized. Decoding then consists of unpacking the Gaussian parameters and rendering the desired view.

efficient encoding scheme. However, the scope of this paper focuses on the rendering step, which is inherently lightweight. Once the parameters are estimated, these are then further quantized and binarized using an arithmetic coder [7, 20].

The next subsection will elaborate on the mathematical formulas present in SMoE which are needed for the rendering of views from SMoE models and of which we will present implementations in Sec. 6.

3.2 Theory

The SMoE framework aims to approximate the underlying function that generates image modality samples. Instead of storing sample grids, this is done using regression. Typically, the goal of regression is to optimally predict a dependent random vector $Y \in \mathbb{R}^q$ from a known random vector $X \in \mathbb{R}^p$. In SMoE, X corresponds to pixel coordinates (i.e., the coordinate space) and Y to the pixel amplitudes (i.e., the color space).

This regression uses a Bayesian variant of the Mixture-of-Experts idea. The joint probability function of the coordinate space X and color space Y is modeled as a multi-modal, multi-variate Gaussian Mixture Model. Each Gaussian kernel then defines a soft-segment and local regressor. The local regressor is defined by a measure of central tendency (e.g. the mean, median, mode) of the conditional function. In this paper, we will limit the case to the mean-estimator. Note that this is the lightest to compute, as it does not rely on the variance of the conditional.

Let us assume $D = \{x^i, y^i\}_{i=1}^N$ to be N pixels to be modeled with coordinates x and amplitudes y:

$$p_{XY}(x,y) = \sum_{j=1}^{K} \pi_j \mathcal{N}(\mu_j, R_j) = \sum_{j=1}^{K} \pi_j \phi_j \tag{1}$$

$$\text{and} \sum_{j=1}^{K} \pi_j = 1, \mu_j = \begin{bmatrix} \mu_{X_j} \\ \mu_{Y_j} \end{bmatrix}, R_j = \begin{bmatrix} R_{X_jX_j} & R_{X_jY_j} \\ R_{Y_jX_j} & R_{Y_jY_j} \end{bmatrix} \tag{2}$$

The parameters of this mixture model with K Gaussian distributions are $\Theta = [\theta_1, \theta_2, \cdots, \theta_K]$, with $\theta_j = (\pi_j, \mu_j, R_j)$, being the population densities, centers, and covariances respectively. The multivariate Gaussian pdf of dimension p is defined as

$$\mathcal{N}_p(x; \mu, R) = \frac{\exp\left(-\frac{1}{2}(x-\mu)^T R^{-1}(x-\mu)\right)}{\sqrt{(2\pi)^p |R|}} \tag{3}$$

Consider a normal pdf of dimension $p+q$, this can be factorized as

$$\mathcal{N}_{p+q}\left(\begin{bmatrix} \mu_X \\ \mu_Y \end{bmatrix}, \sigma^2\right) = \mathcal{N}_q(\mu_{Y|X}, \sigma_Y^2)\mathcal{N}_p(\mu_X, R_{XX})$$

and accordingly for a mixture

$$p_{XY} = \sum_{j=1}^{K} \pi_j \mathcal{N}_{Y|X_j}(m_j(x), \sigma_j^2)\mathcal{N}_{X_j}(\mu_{X_j}, R_{X_jX_j}) \tag{4}$$

with

$$m_j(x) = \mu_{Y_j} + R_{Y_jX_j}R_{X_jX_j}^{-1}(x - \mu_{X_j}), \tag{5}$$

$$\sigma_j^2 = R_{Y_jY_j} - R_{Y_jX_j}R_{X_jX_j}^{-1}R_{X_jY_j} \tag{6}$$

The conditional pdf $Y|X$ is used to derive the regression function [20, 26, 27]:

$$p_Y(Y|X=x) = \sum_{j=1}^{K} w_j(x)\mathcal{N}(x; m_j(x), \sigma_j^2) \tag{7}$$

with mixing weights

$$w_j(x) = \frac{\pi_j \mathcal{N}(x; \mu_{x_j}, R_{X_jX_j})}{\sum_{i=1}^{K} \pi_i \mathcal{N}(x; \mu_{x_i}, R_{X_iX_i})} \tag{8}$$

Note that Eq. 8 corresponds to the normalized exponential or the softmax function frequently used in artificial neural networks and used to determine the support of the model component. The regression of the model is defined as the expected value y given a sample location x through the conditional. From Eq. 7 and 8 follows the regression function $m(x)$:

$$\hat{y} = m(x) = \sum_{j=1}^{K} w_j(x) m_j(x) \tag{9}$$

A signal at location x can be predicted by the weighted sum over all K mixture components (Eq. 9). Every mode in the mixture model is considered as an expert and the experts collaborate towards the definition of the regression function.

4 Parallel Regression Architecture

In this section we outline a general pixel-level reconstruction architecture. In the next section, specific implementations of this architecture are being discussed. The presence of the term "kernels" in SMoE, as well as in GPU architectures creates confusion. In order to avoid this we will refer to "computing kernels" in the case of GPU computing kernels, and "components" when talking about Gaussian steering kernels.

The scope of the proposed architecture is limited to the regression part, once the parameters of the model are unpacked from the bitstream. A full SMoE decoding system consists of entropy decoding, dequantization, and reverse difference coding [20], which are out of the scope of this work.

Given a set of component parameters, the goal of this architecture is to reconstruct 2D views, independent of the dimensionality of the coordinate space. Even when working on higher dimensional modalities, e.g. 4D light fields, the requested views are 2D slices of this higher dimensional coordinate space. This architecture consists of two levels of parallelization. The first level is on block-level, the second level is performed on pixel-level.

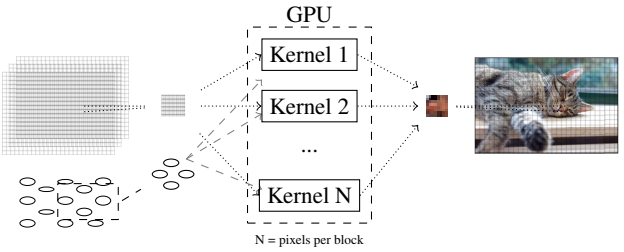


Fig. 7 Block-level parallelism: Each computing kernel is responsible for a block of coordinates. Each block receives a set of Gaussian components that are relevant for this block, which are found by defining a relevance window around this block.

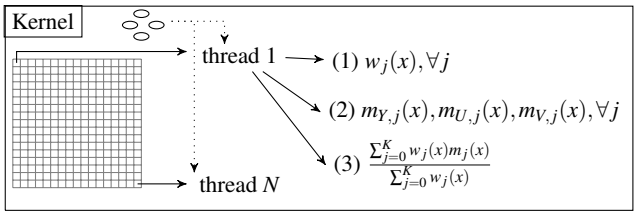


Fig. 8 Pixel-level parallelism: Inside each computing kernel, the kernel dispatches for each sample to be regressed one thread. In each thread a single weight $w_j(x)$ is calculated and three regressors $m_j(x)$ for each color channel. Consequently, in the same thread, the weights are normalized and the weighted sum over each relevant Gaussian component j is calculated. Note that each thread has access to the same set of relevant components and computes the weighted sum over all these components.

4.1 Block-level parallelism

A data point is reconstructed only from the components that have a significant influence on that location. Independent of whether the sample is a 2D pixel location, or whether it is a 4D light field sample location. Due to the competitive nature of training, the amount of components that are responsible for a certain part of the domain are narrowed down. Therefore, the reconstruction for one location tends to be reduced to a weighted sum of a very limited set of components. Only components nearby the reconstructed pixel are considered relevant, as such the memory requirements are constrained and no unnecessary evaluations are performed.

Based on this observation, we perform a computationally cheap and crude subdivision of the 2D coordinate space and the Gaussian components. The coordinate space is divided into rectangular blocks. The assumption is that every block is processed in parallel. However, the regression of the corresponding color amplitudes in the color space Y could rely on components that lie out of the boundaries of this block. This is mitigated by defining a large enough relevance window in X around that block, i.e. each relevant component has a center laying in this relevance window and are thus taken into account during the regression of this block. Fig. 7 illustrates how the coordinate space is divided into blocks and dispatched to separate computing kernel functions.

Alternatively, one could explicitly calculate which components have an influence on this block, which would result in a more fine-grained selection of the components. However, the straightforward method on seeing which components are relevant for which part of the domain is computationally intensive on its own. This requires evaluating the weights (Eq. 8) for each component, thus being $O(K)$ Gaussian evaluations per sample-to-be-reconstructed, which would not be feasible. On the contrary, defining a large enough relevance window based on the Euclidean distance (i.e. taking into account component with centers laying e.g. 32 or 64 pixels outside of the current block), we can avoid any potential overhead in computation time since the calculation of the relevance window always stay in a negligible time range, i.e. 0.1 - 0.3 ms according to our findings.

4.2 Pixel-level parallelism

Within one block, the goal is to reconstruct each pixel independently. As such, for every block calculated in parallel, pixels are reconstructed simultaneously and full parallelization can be claimed. To achieve this, every pixel/block computation will be mapped efficiently on thread blocks which can then be scheduled to run serially or in parallel on a GPU. Another option would have been to operate on component-level. However, the number of pixels per block is always larger than the number of components in our case. As such, it makes more sense to work on a pixel-level basis.

Fig. 8 illustrates the mechanics inside one computing kernel. Each computing kernel takes in a set of sample coordinates and a set of components. The two main functions are: (1) calculating the weights (Eq. 8), and (2) the regressors for each color plane (Eq. 5). Each computing kernel dedicates one thread to each sample which calculate the weights $w_j(x)$ for each component j on that sample, as well as the three color amplitudes $m_j(x)$. Inside the same thread, the summation over each component is done in order to arrive at the reconstructed pixel amplitudes.

Note that the computations inside the threads are non-trivial for GPU implementations. Calculating the weights $w_j(x)$ involves an exponential and the Mahalanobis distance (which includes a matrix inverse). Secondly, a matrix inversion is also present in calculating each color plane. However, with enough care during the implementation we can achieve high speed and precision. The next section discusses the design choices to be taken here.

5 Algorithmic Speed-Precision Optimization

In this section, we will elaborate on the algorithmic decisions that need to be made. Next to speed optimization based on parallelization, we can also achieve speedups by using faster, albeit less precise computations. Experiments that evaluate these decisions are presented in Sec. 7.

Both in the calculation of the weights $w_j(x)$, more specifically the evaluation of the Gaussian pdf (Eq. 3), as well as in the calculation of the expert regressors $m_j(x)$ (Eq. 5), a matrix multiplication with a matrix inverse is present. Multiplying a matrix with a matrix inverse corresponds to solving a system of linear equations. It is well known that multiple computation methods exist that differ in complexity and precision [28, 29].

In order to avoid having tiny coefficients disappearing to zeros, it is common practice in calculating likelihoods to work in the logarithmic domain. The loglikelihood of the Gaussian pdf then boils down to a summation of the log of the constant and the argument of the exponential. The argument of the exponential corresponds to the *Mahalanobis distance*, which is the part that contains the matrix inverse.

Brute-force calculation of the inverse is the fastest method. However, this is also the least precise method as it is sensitive to badly-conditioned matrices. In our implementations, we used closed-form calculations in the GPU implementations [29] as well as in the C++ version, as the inverse() operator in *Eigen* library relies on closed-form expressions for small matrices ($\leq 4 \times 4$) [30].

A mathematical robust way of computing the Mahalanobis distance is to rely on the Cholesky decomposition: $R = L^T L$ (Eq. 10) [29]:

$$(x-\mu)^T R^{-1}(x-\mu) = (x-\mu)^T (L^T L)^{-1}(x-\mu) \quad (10)$$
$$= (x-\mu)^T (L^{-1})^T L^{-1}(x-\mu) \quad (11)$$
$$= (L^{-1}(x-\mu))^T (L^{-1}(x-\mu)) \quad (12)$$

Consequently, it increases the robustness as the eigenvalues of L are square roots of the eigenvalues of R. If the eigenvalues are close to zero (and thus dangerous for computational accuracy), then the square root ensures that the new eigenvalues are at least larger. Consequently, it is clear that given the step in Eq. 11, the computation of the argument boils down to calculating $L^{-1}(x-\mu)$. This still requires solving a system of linear equations and L can still be badly-conditioned. In order to maximize numerical stability, a robust method (although slower) should be used, e.g. QR-decomposition or Singular Value Decomposition (SVD) decomposition [29]. In our implementation, we chose for QR-decomposition as it is robust but still faster than SVD. The QR decomposition issues a transform onto an orthogonal axis, which ensures the resulting inverse to be more stable. As such, numerical drift is reduced compared to immediately computing the inverse.

In order to calculate the linear regressor $m_j(x)$ for each component (Eq. 5), the following linear system needs to be solved: $R_{YX} R_{XX}^{-1}(x-\mu)$. In this case, we can choose again to use brute-force inverse or a slower-but-robust method (e.g. QR-decomposition).

To conclude, we state that there are design choices to be made that potentially influence the visual reconstruction fidelity. A more robust method of solving a linear system (i.e. using QR decomposition) also increases computation time. If however, the matrices to be inversed are well conditioned, then such a decomposition results in unnecessary overhead. We thus define a *slow* mode, one using robust computations (i.e. using QR decomposition), and a *fast* mode, one using straightforward but potentially sensitive computations.

6 Implementation Details

This section describes the implementation details of parallel hardware-accelerated SMoE decoding in two GPU-accelerated languages: Quasar and OpenCL. We discuss our technique to map the algorithm efficiently on a GPU, and offer a number of optimization strategies. As discussed in the previous section, the performance behavior of the high and low numerical accuracy versions are presented. Finally, a brief discussion of the single-threaded C++ CPU version is included for completeness and reference.

6.1 Overview

Many research fields have progressed faster due to the increasing numerical computation power that massive multi-core processors in GPUs and in current many-core CPUs bring to the table. The performance gain is generally achieved by structuring multi-threaded computations so that parallelism and data locality are exploited. In turn, the computational challenges that researchers aspire to conquer have become larger and more complex.

The low-level *Compute Unified Device Architecture* (CUDA) language [31], which works exclusively on NVIDIA graphics hardware, consistently outpaced all other parallel processing platforms in terms of performance, features and popularity. Similar to CUDA in terms of language and performance while at the same time also retaining cross-platform support for parallel programming on heterogeneous systems, is the OpenCL standard by the Khronos Group [11, 32]. For this reason (hardware independence), we opted for an OpenCL implementation.

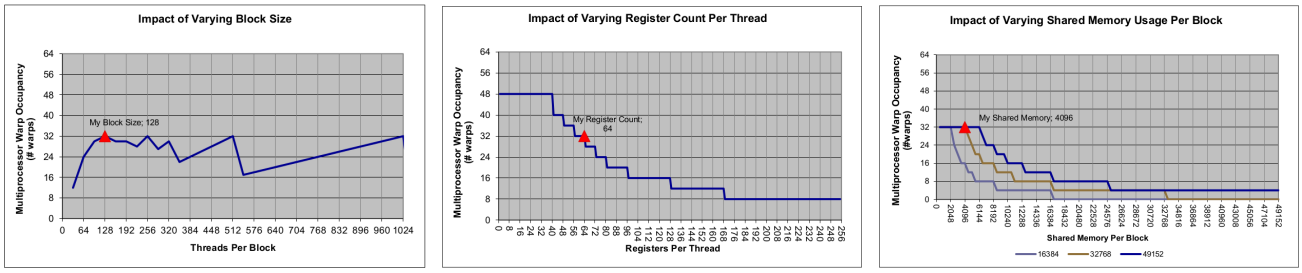
The high-level parallel programming framework *Quasar* hides implementation complexities so that development can focus entirely on the algorithmic part [12, 13]. The Quasar runtime system consists of four major components: a memory manager, a scheduler, a load-balancer, and a device back-end. Each of these components allows code-optimizations at different stages of the pipeline. The device back-end then communicates with the underlying hardware through CUDA or OpenCL. These features are used in a performance comparison of our SMoE algorithm using a GPU-parallel CUDA approach. Hence, Quasar is used as a high-level development tool to prove the feasibility of parallelizing SMoE on massively parallel hardware without expert knowledge. To verify our results, a native OpenCL implementation of SMoE image and light field rendering at real-time frame rates was also developed. With this approach we hope to cover an adequate number of parallel implementations ranging from high-level language to low-level native interface approaches and provide a comprehensive overview. In Sec. 6.2, we discuss implementations details for Quasar, and in Sec. 6.3, we provide details about the OpenCL version. Finally, experimental evaluation results are discussed in Sec. 7.

6.2 Quasar implementation

Following the optimization guide that is distributed with the Quasar platform [35], our presented solution avoids the use of dynamic memory such as described in [36] and dynamic parallelism, and instead operates on fixed-size buffers and vectors. As it has been described in previous sections, the SMoE algorithm requires the inverse of the covariance matrix R_j to compute each component weight, and again to compute the final pixel color reconstruction. A Cholesky decomposition and/or a QR decomposition for solving a set of linear equations can be used to increase numerical stability and precision of the computations, at the cost of wall-clock performance. Hence, a number of closed-form functions for fixed-size inputs are developed for calculating the inverse of a square matrix, the Cholesky decomposition and QR decompositions for that matrix. To execute functions as *device functions* and maximize the use of vector registers for optimal performance, matrix vectorization is achieved by using vectorized memory access. The vectorized approach decreases the use of dynamic memory allocations in Quasar, reducing both buffer synchronization and addressing. To further improve performance, in the Quasar approach all output image blocks are processed simultaneously using one big GPU kernel. More specifically this was achieved by using a 4D parallel loop (over the three color channels, 2D block index, and Gaussian components).

Quasar uses various optimization techniques such as automatic generation of kernel functions for matrix expressions, automatic parallelization of for-loops, register allocation optimization and reductions. A number of additional practices were deployed for boosting performance and/or maintaining stability while operating on the GPU. An example is uninitialized memory allocation. Using the function *zeros* or *ones* for allocating memory is a common practice in high-level programming. However, for those functions it takes an extra step for clearing the memory associated to the matrix to be allocated. Even if this step takes no more than 50 μs, it can still be avoided by using the Quasar function *uninit* when that matrix is going to be overwritten by a following operation. Another practice we used is kernel boundary check omission. Kernel arguments are being checked for index-out-of-bounds by the kernel boundary check transform. The boundary check was omitted with the use of the argument modifier *unchecked* thereby assuming that the index never goes out of bounds. Vectors with a known size smaller than 32, allowed us to store them directly into the registers of the device as fixed-length vectors (vec4, vec8, vec16, etc.). Accessing registers is significantly faster than accessing global memory and leads to computational gain.

In the implementation of the SMoE algorithm in Quasar, two major optimizations have lead to a significant performance increase: (1) data layout optimization (placing all model parameters inside one data matrix for which the number of columns is aligned to the cache line size of the GPU and (2) use of *shared memory* of the GPU. *Shared memory* is on-chip memory that has a lower latency (about a factor 100) than global memory. Shared memory is shared between all threads within one block of the GPU and limited in size (typically, 48KB per block). To take optimal use of shared memory, we made trade-offs between the number of registers, the number of threads and the amount of shared memory used by the kernel on the one hand and between global memory access and recalculation on the other hand. Rather than precalculating the Cholesky decomposition of the model covariance parameters (because each decomposition is used sev-



(a) Impact of block size (b) Impact of number of registers per thread (c) Impact of shared memory usage per block

Fig. 9 Impact of GPU occupancies when varying (9a) GPU block sizes, (9b) register count per thread and (9c) shared memory usage per thread. All measurements are extracted from the the official CUDA Occupancy Calculator [33] (property of NVIDIA) for the OpenCL SMoE implementation. As it can be seen from a, b and c, there is only room for improvement when reducing the number of registers to achieve higher occupancy. The achieved occupancy per multiprocessor is in this case at 50%. However, note here that higher occupancy does not necessarily mean better performance [34].

eral times), we compute several Cholesky decompositions on-the-fly in parallel and then broadcast the results to all the GPU threads using shared memory. Consequently, several recomputations of Cholesky decompositions are being performed, but because the recomputation is rather fast and makes optimal use of the shared memory of the GPU, this approach hides the global memory latency, leading to a significantly improved computational performance compared to an implementation that is not using shared memory. In Quasar, shared memory allocation is obtained by calling the function *shared*, synchronization of threads in a GPU block is obtained using the keyword *syncthreads(block)*.

6.3 OpenCL implementation

To estimate the feasibility of 2D and 4D SMoE reconstruction visualization for a specific hardware platform, we tested a native parallel GPU version of the SMoE algorithm that combines insights from the Quasar implementation with common *General Purpose GPU* (GPGPU) optimization techniques and considerations [37]. As each numerical operation attributes numerical precision loss that may result in PSNR error, native implementations offer a level of control that is required to understand optimal SMoE model configurations as well as hardware and software requirements to counter the propagation of error.

The Khronos' OpenCL API offers both coarse and fine-grained hardware parallelism and is primarily chosen because of its extensive portability proposition [32]. In the next paragraphs, we describe our algorithm in OpenCL's terminology applied to NVIDIA architectures.

In contrast to the Quasar implementation, *shared memory* was not used to compute matrix inverses in the OpenCL version. Although promising in terms of benchmarking, we also did not use the *clDataTransfer* communication techniques proposed by Takizawa et al in [38]. Instead, the Gaussian kernel buffer and index buffer are pinned to the host, i.e. they cannot be paged out while the driver synchronizes using *Direct Memory Access* (DMA) transfers to the GPU. Using explicit *map* and *un-map* functions, all covariance and domain information (i.e. the pixel coordinates to be reconstructed) is uploaded to the GPU in blocks of 4096 values before kernel execution starts. Image output is written to a shared and resident OpenGL texture buffer for immediate rendering, hereby avoiding host round-trip delay. The pixel-parallel nature of the SMoE algorithm does not require the use of (dynamic) memory allocations, but further research is required to reduce the computational complexity per kernel by using more memory intensive techniques, for example by improving data locality using hash-tables. All pixels evaluate k relevant components in parallel.

GPU *occupancy* is a measure that expresses the efficiency of the GPU hardware scheduler as it distributes computing tasks across hardware threads. Occupancy improves when *branching divergence* within a *warp* (a group of 32 threads or a multiple thereof in our experiments) is minimized. We selected work groups of 16×8 threads that each share 48 KB of memory. In Fig. 9, we visualize the theoretical occupancy of this OpenCL implementation when calculated using NVIDIA's CUDA Occupancy Calculator [33]. According to the selected GPU block sizes, it can be seen that a 50% occupancy has been reached with the only bottleneck being the number of registers per thread. Although higher occupancy does not necessarily mean increased performance [34], further research can be done on this implementation for reaching the optimal occupancy. Depending on the Gaussian component dimension and the use of registers in the GPU kernel computation, more than one Gaussian component can be stored in faster (local) shared memory, allowing pre-fetching of the data and avoiding global memory addressing penalties. The amount of shared memory that is required must not exceed hardware capacity in order to avoid warp re-scheduling operations that result in suboptimal performance. Additionally, using less registers increases performance and reduces instruction latency. Therefore, 2D Gaussian components are trimmed from the original 31 values to 18 *unique* floating point values. The same trimming operation results in 33 *unique* values for a 4D Gaussian compo-

nent. Our 2D experiment simultaneously loads 11 Gaussian 2D components into 256 shared floating point variables per work group, allowing each pixel to compute the weighted contribution of these 11 Gaussian 2D components in one iteration step. The relevant set of Gaussian components per block is loaded from an index table. After all relevant Gaussian components have contributed to each color channel, the pixel color is reconstructed. For a single compute domain block of 512×512 threads (pixels), it was empirically determined that full image reconstruction and visualization is feasible in under 16 ms when approximately 265 relevant components per block are used. This includes 2.2 ms render-framework overhead. At the CPU side, the OpenCL *clEnqueueNDRangeKernel*() function is used to subdivide the compute domain block of maximum size of 512×512 into hardware workgroups. The reconstruction of larger images is achieved by subdividing them into compute domain blocks and repeating the procedure for each compute domain block.

Analogous to our Quasar experiments, all matrix operations are implemented in OpenCL as closed form functions and vectorized using native OpenCL vector primitive types such as *vec*2, *vec*4 and *vec*16, and fast native approximation intrinsics such as *native_sqrt*(), *native_log*() and *native_exp*(). Timing results are produced for the complete reconstruction of a single block for all block-sizes, including each Gaussian weight computation and the final pixel reconstruction of the SMoE algorithm. We measured an accurate version that uses a combination of a QR matrix decomposition and a Cholesky-based inversion, and a faster version that uses unrolled loop matrix inversion in order to test the sensitivity of the SMoE algorithm for accuracy and performance. Some computations are simplified by reducing the number of instructions, i.e. the computation of the final pixel reconstruction does not require two full matrix multiplication expansions. Pointer dereferencing is not used to avoid GPU hardware addressing logic.

The block-sizes for each dataset are multiples of the preferred hardware requirements and workgroup sizes, and ideal for further subdivision in hardware workgroups. For visualization applications however, care must be taken to convert OpenCL's spatial interpretation of local and global thread id's (starting from the lower-right corner) in the 2D compute domain to OpenGL's interpretation that samples textures starting from the top-left corner. Furthermore, the original image in Matlab has the Y axis pointing upwards, while the data-set stores blocks using a Y, X coordinate indexing scheme. Additionally, the input domain X samples the center of pixels, i.e. not at (0, 0) for the 2D case. To minimize performance impact, index translation is resolved before Gaussian kernel-weight computations start. If the image resolution is not a multiple of the block-size dimension, the compute domain is padded with one extra compute domain block to streamline hardware processing so that all pixels are reconstructed correctly.

The OpenCL API offers an event-based device timer that reveals how asynchronous GPGPU execution behaves by measuring and reporting when data transfer and processing operations are queued, started and terminated by the driver. Our Titan X device reports a device timer granularity of 1000 nanoseconds. Buffer upload transfer times (map and un-map operations) for the covariance matrices and relevance index tables, the kernel executions and, if applicable, buffer download transfer times (map and un-map operations again) for partial results such as the reconstructed color channel values and weights are benchmarked.

Note that this implementation was built assuming to reconstruct the whole image, and not a single block. As such, instead of reducing the compute domain when measuring performance benchmarks for single blocks, the list of relevant Gaussian components for all other blocks is left empty. In this way, exactly one block performs at least one or more Gaussian component weight computations and pixel reconstructions, yielding an accurate benchmark result for that block. Small differences in the data transfer latency of blocks of the same block-size are thus likely caused by the device and driver specific implementation details, and cannot be attributed to the different numbers of relevant Gaussian components for each block.

6.4 C++ single-thread implementation

A straightforward C++ version of the algorithm was implemented using the library *Eigen* (version 3.3) [30]. It follows the same block-level subdivision, although it does the pixel-level parallelism differently compared to the outlined GPU architecture. When working on the CPU, it is much more beneficial to do the computations on large matrices due to efficient vectorization in modern hardware. Point-per-point evaluation would result in high computational overhead. Note that it is possible to implement a multi-threaded CPU version, however, the achieved speedup would maximally be $\times T$, T being the number of CPU threads.

A fast implementation based on the inverse was done using the built-in matrix inversion functions in Eigen. For calculating the loglikelihood, the Cholesky decomposition is used as in Eq. 12. Otherwise the left-hand side of Eq. 12 would result in a *NxN* matrix, because in this version x is a N-dimensional vector. Without the Cholesky factorization the matrix multiplication would thus result in huge temporary memory storage, which we want to avoid. The slow implementation uses the Householder rank-revealing QR decomposition of a matrix with column-pivoting (using the Eigen-function *ColPivHouseholderQR*) for both the Mahalanobis distance after the Cholesky decomposition, as well as for the calculation of the linear regressors $m_j(x)$.

(a) 2D *Lena* ($K = 2095$) (b) 2D *Car* ($K = 21802$) [39] (c) 4D *Bikes* ($K = 8960$) [23] (d) 4D *Friends* ($K = 10080$) [23]

Fig. 10 For the experiments four SMoE models were used that originate from four sources: two 2D images (*Lena* & *Car*) and two 4D light fields (*Bikes* & *Friends*), with various K, i.e. the number of components. The reconstruction of the models are shown here. The two source images had a 512x512 resolution. The source light field had 13x13 angular resolution and 434x626 spatial resolution.

7 Experimental Evaluation

7.1 Introduction

The goal of this section is to provide experimental evaluation of the SMoE parallel implementations in terms of speed and visual fidelity. Four experiments are presented. Firstly, the algorithmic speed-precision trade-offs presented in Sec. 5 are evaluated. Secondly, the performance of all implementations are evaluated on single blocks for a range of block-sizes in order to validate the pixel-level parallelism from Sec. 4.2. Thirdly, we compare the speed of the fastest implementation on a high-end system versus a consumer laptop. Finally, we evaluate the frame rate of a high-detailed 1080p and 4K image by simulating the block-level parallelism from Sec. 4.1.

7.2 Dataset

We conducted the experiments on two different image modalities, i.e. 2D color images and two static 4D light fields. For the first scenario, the test set consists of two 2D color images (*Lena* and a crop from *Car* of resolution 512x512 [39]). For the light field case, the static 4D light field *Bikes* and *Friends* [23] were used. These images were modeled using the SMoE approach with various model sizes. The reconstructions of these models are illustrated in Fig. 10.

In order to simulate the output of the block-level parallelism from Sec. 4.1 for different chosen block-sizes, we output the pixel-coordinates and the relevant Gaussian components for those pixel-coordinates using block-sizes of $[16, 32, 64, 128, 256]$ as in Fig. 7. Smaller block-sizes result in blocks containing less pixel locations and less relevant components, however, it increases the number of relevance lookups. The relevance of the Gaussian components are decided by the relevance window. The less components a model has, the larger the relevance window should be taken in order to be sure to capture all relevant components. Therefore, for *Lena* ($K = 2095$), *Car* ($K = 21802$), *Bikes* ($K = 8960$) and *Friends* ($K = 10080$), the relevance windows used were respectively the blocksize plus an outer border of $64, 16, 32$, and 32.

Using the high-precision MATLAB implementation we output for each block for each image for every block-size, the weights $w_j(x)$ and the three regressors $m_{Y,j}(x), m_{U,j}(x), m_{V,j}(x)$, and the final regression for each color channel with a precision of 20 digits (20 decimal digits per color channel in ASCII format). These values are then used to compare the fidelity of the parallel implementations in the following experiments. Note that the color channels are normalized to $[0-1]$.

7.3 Hardware

For the experiments two setups have been tested, one aiming for performance and one for availability. For the first case, a machine operating with an *Intel® Core™ i7 - 5960X CPU @3.00 GHz* and assisted with a Maxwell-based *NVIDIA GeForce GTX Titan X* graphic device has been employed. Aside from standard watercooling hardware for all processors, no special overclocking strategies are used. For the second case, we used a commodity laptop with an *Intel® Core™ i7 - 6700HQ CPU @2.60 GHz* assisted by an *NVIDIA GeForce GTX 960M* graphic device. While the *Titan X* machine consists of 3072 CUDA cores, the *960M* consists of a more limited number of 640 CUDA cores.

7.4 Algorithmic precision evaluation

In this experiment, we compare the influence of the algorithmic choices presented in Sec. 5, using the dataset created by the high-precision implementation using MATLAB as in Sec. 7.2. The goal is to evaluate how much visual loss a faster, but less robust scheme produces.

Precision has been measured in terms of *Mean Squared Error* (MSE) between each parallel coder and ground truth. The most commonly supported precision in modern devices is 8-bit (256 color levels) per color channel. There are usually three color channels, so that makes a color depth of 24 bits-per-pixel (bpp). Modern codecs support precisions of

Table 1 Precision comparison for OpenCL fast vs. slow implementation for *Car* at block size of 64.

	Luminance MSE	Max pixel error	Bitdepth fidelity
Fast	2.3891e-05	4.2367e-06	14-bit
Slow	2.4083e-05	4.5229e-06	14-bit

10- and 12-bits per channel (1024 colors and 30 bpp and 4096 colors and 36 bpp). Therefore, one of the goals is to validate the bit-depth fidelity of our implementations for 8, 10, and 12 bits and report the highest precision that can be delivered. Furthermore, the precision loss for each implementation must stay as limited as possible, and the precision difference between implementations must range within reasonable limits, especially regarding the reconstructed pixel intensities.

We measured the MSE between the ground truth and our implemented parallel rendering frameworks, and observed that it stays within a reasonable low range. An example for the dataset *Car* is given in Table 1. Interestingly, we noticed that for both our fast and slow versions (as defined in Sec. 5) the MSE and the maximum pixel error stays not only particularly low but it is almost identical at all cases.

Finally, we quantified the number of reconstructed pixels that satisfy the rule $MSE < 1/2^b$, where b is $[8,10,12]$ for 8-, 10-, and 12-bits precision respectively. In all cases, the number of pixels which do not satisfy that rule is zero. The highest bit-depth precision we can reach without any losses is 14-bit, while for the case of 16-bit a very limited amount of pixels exceeds the above described limit (< 10 at all cases). We can conclude that for the given datasets, it is valid to proceed to time evaluation using only the fast-and-precise-enough version of the algorithm based on the matrix inverse, instead of the QR-decomposition. The sufficiently conditioned covariance matrices can be explained by the regularization during modeling, which is done in order to prevent singularities in the covariance matrices [5].

7.5 Implementation speed-evaluation for single blocks

Since the above experiment indicates that for the chosen datasets the fast implementation (i.e. relying on the inversed matrix) shows high enough precision, we perform the forthcoming experiments using this mode. In this experiment, we evaluated our results in terms of speed, as illustrated in Fig. 11. We also compare to Quasar implementations, one with and one without the use of shared memory (see Section 6.2). The total timings for a single-block reconstruction were measured on the high-end *NVIDIA GeForce GTX TITAN X* machine. Note here, that the timings were acquired by using the Quasar built-in function for time measurements (which internally relies on CUDA events), while for the OpenCL implementation we derived the timings using *QueryPerformanceCounter()* on each `cl_kernel` execution round-trip.

Table 2 Indicative total timings for a block (128x128 pixels) in seconds and speedup factors when compared to the C++ reference decoder. All timings were taken when working on a four *Nvidia GeForce GTX TITAN X* GPU machine.

Dataset	C++	Quasar	Quasar (SM)	OpenCL
Lena 2D	1.2s	0.0008s	0.0003s	0.0005s
Car 2D	3.5s	0.0021s	0.0007s	0.0013s
Bikes 4D	3.5s	0.0120s	0.0036s	0.0029s
Friends 4D	3.9s	0.0131s	0.0041s	0.0043s

Table 3 Data transfers average timings in Quasar and OpenCL for dataset *Car*. It can be seen that data transfers are in the range of μ *s* and they can be considered almost negligible. Average timings for datasets *Lena*, *Bikes* and *Friends* are in the same range. Data transfers are optimized automatically by the Quasar data transfer optimizer. OpenCL writes its image output to a shared and resident OpenGL texture buffer for immediate rendering and avoids a host round-trip delay, therefore we only measure the time from CPU $\rightarrow$ GPU.

	Quasar	Quasar	OpenCL
Block size	CPU $\rightarrow$ GPU	GPU $\rightarrow$ CPU	CPU $\rightarrow$ GPU
16	$67\mu s$	$92\mu s$	$0.3\mu s$
32	$65\mu s$	$96\mu s$	$1.1\mu s$
64	$69\mu s$	$113\mu s$	$4.4\mu s$
128	$83\mu s$	$119\mu s$	$17\mu s$
256	$131\mu s$	$172\mu s$	$71\mu s$

The C++ reference decoder is used as a comparison measure. As an example, for block sizes of 128×128, the high-level Quasar implementation reached speedup factors in the range of $\times 1500 - 1666$ for 2D images and $\times 291 - 297$ for 4D light fields, while the Quasar implementation using shared memory reached speedup factors up within $\times 4000 - 5000$ and $\times 951 - 972$ for images and light fields respectively. The OpenCL implementation reached speedups in the range of $\times 2400 - 2692$ and $\times 906 - 1206$ for 2D images and 4D light fields respectively (see Table 2).

Data-transfer timings are shown in Table 3. The data-transfer times are close to negligible. As such, we only include the timings for the computations on the GPU. The OpenCL version still reduces the data transfer substantially, likely caused by numerous data optimizations and trimmings as discussed in Sec. 6.3. Furthermore, it also writes directly to the OpenGL texture buffers so there is no GPU-CPU column. Fig. 11, shows that there is an optimum when looking at the average time per pixel. This means that sharing the same set of components for a set of pixels is optimum for block-sizes around 64. This can be used to lower data transfer bandwidth to the GPU and storage. However, it is always beneficial for rendering speed per pixel to have less compo-

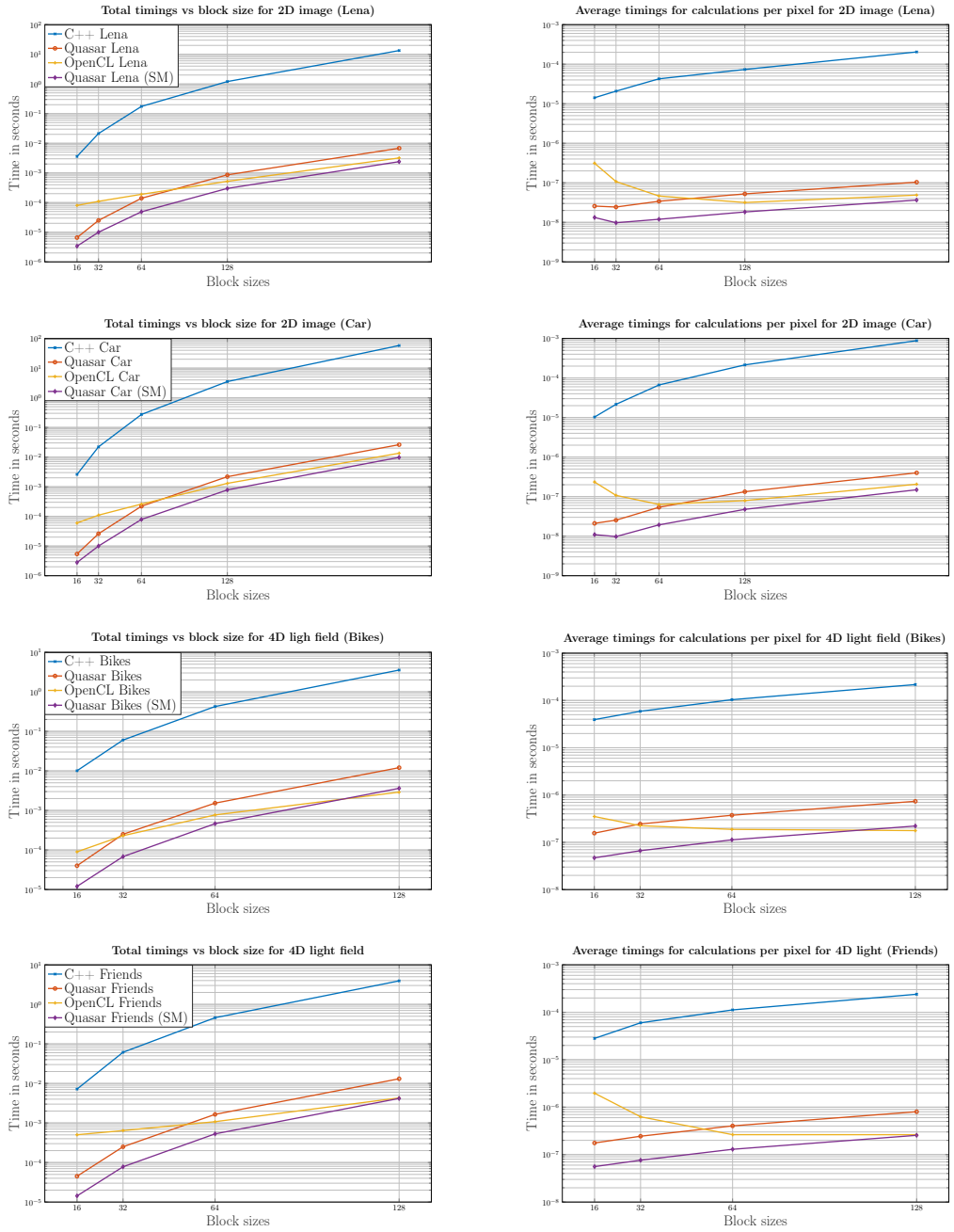


Fig. 11 Time evaluation for *Lena*, *Car*, *Bikes*, and *Friends* from top to bottom respectively for the *C++*, *Quasar*, *OpenCL* and *Quasar SM* (Quasar implementation using shared memory). Left: total timings in seconds for different block sizes. Right: Average time in seconds for calculations per pixel. Note that the timings here are for a single block reconstruction with the assumption that blocks are reconstructed in parallel. While for the total timings the time per block increases when the block size increases, for the average time per pixel we can see that in certain cases different block sizes are the best-fit for different data sets. Logarithmic scale has been used for better comparison.

nents to evaluate per pixel, so in practice, lower block-sizes will result in higher frame rates if data transfer is not the bottleneck.

Note that the dimensionality of the model seems to not have a high influence on rendering speed. The amount of components does however have a severe impact, i.e. a 2D model with high K will render slower than a 4D model with smaller K.

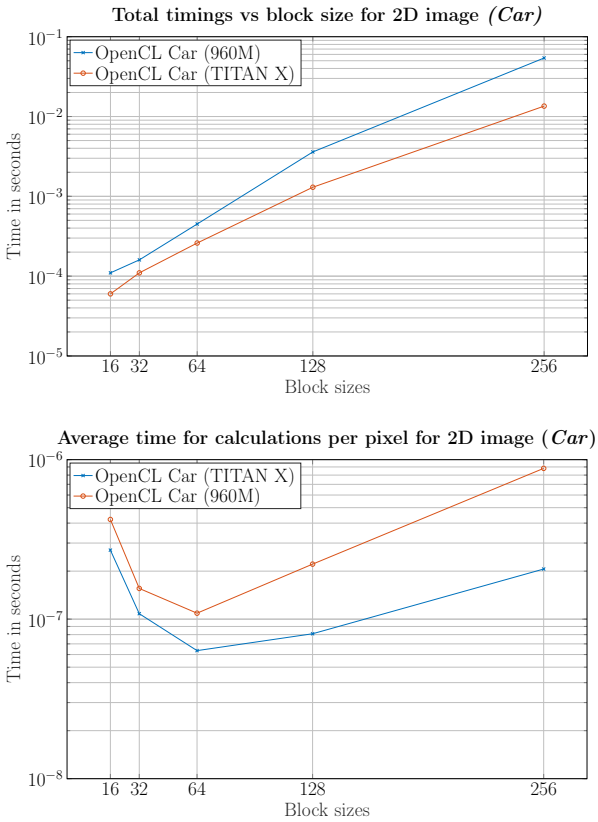


Fig. 12 Time evaluation for *Car* using an NVIDIA GTX TITAN X vs an NVIDIA GTX 960M. Top: total timings in seconds for different block sizes. Down: Average time in seconds for calculations per pixel. Note that the timings here are for a single block reconstruction with the assumption that blocks are reconstructed in parallel. It can be seen that even with a less powerful, more common GPU, the timings can stay within the needed real-time constraints (under 40 ms).

7.6 Hardware complexity influence for single blocks

In this experiment, we tested the pixel-parallel SMoE renderer in two different hardware setups. The first setup was a laptop using an *NVIDIA GeForce GTX 960M* and second setup was a desktop machine using a *NVIDIA GeForce GTX TITAN X*. Due to its popularity and accessibility we chose to use the OpenCL implementation for this demonstration. Nonetheless, for the rendering scenarios using Quasar on a laptop device, the results are similar and corresponding to Sec. 7.5.

Fig. 12 shows that in terms of reconstructing a single block, the consumer-grade hardware is able to achieve real-time rendering as well. However, it is expected that the throughput of number of blocks that can be rendered simultaneously differs as shown in the experiment in the subsequent section (Sec. 7.7).

Table 4 Rendering speeds for a single frame for 1080p and 4K resolutions, as well as the average Gaussian kernels-per-block (kpb). As the models are simulated by repeating blocks from the dataset, the overhead of the block-level process (i.e. relevance look-up) is not included. As such, smaller blocks lead to the fastest rendering speeds as they yield less Gaussian evaluations per pixel. For the 960M machine, some results lack due to driver crashes.

		Titan X	Titan X	960M	960M
Block size	kpb	1080p	4K	1080p	4K
16	100.9	11.7ms	44.4ms	51.7ms	155.1ms
32	177.95	18.8ms	69.6ms	69.1ms	269.6ms
64	398.59	36.1ms	141.6ms	151.3ms	816.7ms
128	1087.3	92.7ms	365.5ms	483.4ms	1634.8ms
256	3665.5	309.6ms	2229.6ms	1750.1ms	-

7.7 Simulation of high-detailed 1080p and 4K images

In this experiment, we simulate a high definition image by copying over the *Car* model horizontally and vertically in order to arrive at two models that simulate sources of resolutions HD/1080p (1920x1080) and 4K UHD-1(3840x2160). As such, we mimic a model that was created from a high resolution source. *Car* was chosen as it has the highest number of Gaussian kernels, and we have shown that the number of components is the most determining parameter for rendering speed. The simulated models have respectively $K = 169401$ and $K = 686763$ Gaussian kernels. As such, we can compare the influence of the block-sizes on a real-world scenario.

Table 4 shows the results for the two machines in consideration. It is clear that the weaker machine is not able to achieve similar throughput and does not deliver timings that are acceptable in terms of real-time rendering, although it comes close for 1080p with 20 fps. However, the machine that contains the Titan X GPU, does provide 85.5 fps and 22.5 fps for respectively 1080p and 4K.

Furthermore, we find that lower block-sizes result in the highest frames-per-second. This is caused by the fact that lower block-sizes cause less component evaluations per pixel. Consequently, we conclude that in this implementation it is more important to have a low amount of component evaluations per pixel, rather than having optimal component sharing for data transfer. Note however that timings did not include the relevance-selection process (Sec. 4.1) as this was already done when constructing the dataset (Sec. 7.2). The smaller the blocks, the higher the number of relevance lookups as the number of blocks increases. We found that while constructing the dataset in MATLAB the lookup process accounted for 0.1-0.3ms per block, this could thus have an impact. However, this lookup process can also be further optimized and even straightforwardly be parallelized on a per-kernel basis.

8 Conclusions and Future Work

In this paper we have proven the viability of pixel-level rendering of SMoE models. We presented a general high-level architecture for rendering SMoE models. Next, we evaluated the algorithmic design choices within the renderer. And finally presented two GPU implementations that differ in level of abstraction. We have shown that the OpenCL and Quasar implementations achieve speedup factors of up to $\times 5000$ compared to a single-threaded C++ version. Furthermore, we have shown that the dimensionality of the model (2D image or 4D light field) is less of a determinant factor than the amount of components in the model. Finally, we have simulated high-detailed models of respectively 169401 and 686763 Gaussian components and rendered them at 1080p and at 4K resolutions. We have shown that rendering at frame rates of respectively 85.5 fps and 22.5 fps is possible given appropriate hardware.

It is clear that the amount of Gaussian evaluations per pixel is best kept as low as possible. Smaller block-sizes at block-level ensure this. However, the overhead of the component relevance process at block-level could produce significant overhead. Future work consists of evaluating fast selection strategies and implementations. Gaussian component numbers can also be reduced by improving SMoE modeling efficiency algorithmically. Using more optimized modeling algorithms, we would need to achieve the same level of visual quality with less Gaussian components. On implementation level, the modeling still holds challenges. However, as the rendering step corresponds to the E-step in the EM-algorithm, the same ideas can be used to develop a highly parallelized SMoE modeler. Especially keeping in mind the extreme sample sets that will be present in the envisioned 6DoF content.

References

1. Ihrke I, Restrepo J, Mignard-Debise L (2016) Principles of Light Field Imaging: Briefly revisiting 25 years of research. IEEE Signal Processing Magazine 33(5):59–69, DOI 10.1109/MSP.2016.2582220
2. Adelson E, Bergen J (1991) The plenoptic function and the elements of early vision. Computational Models of Visual Processing (MIT Press) pp 3–20
3. Levoy M, Hanrahan P (1996) Light field rendering. In: Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '96, ACM Press, New York, New York, USA, pp 31–42, DOI 10.1145/237170.237199
4. Wu G, Masia B, Jarabo A, Zhang Y, Wang L, Dai Q, Chai T, Liu Y (2017) Light Field Image Processing: An Overview. IEEE Journal of Selected Topics in Signal Processing pp 1–1, DOI 10.1109/JSTSP.2017.2747126
5. Verhack R, Sikora T, Lange L, Van Wallendael G, Lambert P (2016) A universal image coding approach using sparse mixture-of-experts regression. In: Proceedings of the IEEE International Conference on Image Processing (ICIP), pp 2142–2146
6. Lange L, Verhack R, Sikora T (2016) Video representation and coding using a sparse steered mixture-of-experts network. In: Picture Coding Symposium (PCS), pp 1–5
7. Verhack R, Sikora T, Lange L, Jongebloed R, Van Wallendael G, Lambert P (2017) Steered mixture-of-experts for light field coding, depth estimation, and processing. In: Proceedings of the IEEE Conference on Multimedia and Expo (ICME), pp 1183–1188
8. Sullivan GJ, Ohm JR, Han WJ, Wiegand T (2012) Overview of the High Efficiency Video Coding (HEVC) Standard. IEEE Transactions on Circuits and Systems for Video Technology 22(12):1649–1668, DOI 10.1109/TCSVT.2012.2221191
9. Domanski M, Stankiewicz O, Wegner K, Grajek T (2017) Immersive visual media MPEG-I: 360 video, virtual navigation and beyond. In: 2017 International Conference on Systems, Signals and Image Processing (IWSSIP), IEEE, pp 1–9, DOI 10.1109/IWSSIP.2017.7965623
10. Hinds AT, Doyen D, Carballeira P (2017) Toward the realization of six degrees-of-freedom with compressed light fields. In: 2017 IEEE International Conference on Multimedia and Expo (ICME), IEEE, pp 1171–1176, DOI 10.1109/ICME.2017.8019543
11. Stone JE, Gohara D, Shi G (2010) OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science & Engineering 12(3):66–73, DOI 10.1109/MCSE.2010.69
12. Goossens B, De Vylder J, Philips W (2014) Quasar - a new heterogeneous programming framework for image and video processing algorithms on cpu and gpu. In: Proceedings of the IEEE International Conference on Image Processing (ICIP), pp 2183–2185
13. Goossens B (2018) Dataflow Management, Dynamic Load Balancing and Concurrent Processing for Real-time Embedded Vision Applications using Quasar. International Journal of Circuit Theory and Applications, Special Issue of Computational Image Sensors and Smart Camera Hardware
14. MATLAB (2015) version 8.6.0 (R2015b). The MathWorks Inc., Natick, Massachusetts
15. Smola A, Schölkopf B (2004) A Tutorial on Support Vector Regression. Statistics and computing 14(3):199–222
16. Broomhead DS, Lowe D (1988) Radial basis functions, multi-variable functional interpolation and adaptive networks. Tech. rep., DTIC Document
17. Altinigneli MC, Plant C, Böhm C (2013) Massively parallel expectation maximization using graphics processing units. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, pp 838–846

18. Moon T (1996) The Expectation-Maximization Algorithm. IEEE Signal Processing Magazine 13(6):47–60, DOI 10.1109/79.543975
19. Kumar NSLP, Satoor S, Buck I (2009) Fast Parallel Expectation Maximization for Gaussian Mixture Models on GPUs Using CUDA. In: 2009 11th IEEE International Conference on High Performance Computing and Communications, IEEE, pp 103–109, DOI 10.1109/HPCC.2009.45, URL http://ieeexplore.ieee.org/document/5166982/
20. Verhack R, Sikora T, Lange L, Van Wallendael G, Lambert P (2018) Steered Mixture-of-Experts for 4-D Light Field Approximation, Coding, and Description. Submitted to IEEE Transactions on Multimedia
21. Yuksel SE, Wilson JN, Gader PD (2012) Twenty Years of Mixture of Experts. IEEE Transactions on Neural Networks and Learning Systems 23(8):1177–1193, DOI 10.1109/TNNLS.2012.2200299
22. Viola I, Rerabek M, Bruylants T, Schelkens P, Pereira F, Ebrahimi T (2016) Objective and subjective evaluation of light field image compression algorithms. In: 32nd Picture Coding Symposium, EPFL-CONF-221601
23. Rerabek M, Ebrahimi T (2016) New light field image dataset. In: 8th International Conference on Quality of Multimedia Experience (QoMEX), EPFL-CONF-218363
24. Tok M, Jongebloed R, Lange L, Bochinski E, Sikora T (2018) An MSE approach for training and coding steered Mixtures of Experts. In: Accepted for publication in Picture Coding Symposium (PCS '18), San Francisco, California USA
25. Bochinski E, Jongebloed R, Tok M, Sikora T (2018) Regularized Gradient Descent Training of Steered Mixture of Experts for Sparse Image Representation. In: Accepted for publication in IEEE International Conference on Image Processing (ICIP '18), Athens, Greece
26. Sung H (2004) Gaussian Mixture Regression and Classification. PhD thesis, Rice University
27. Bugmann G (1998) Normalized Gaussian Radial Basis Function networks. Neurocomputing 20(1-3):97–110, DOI 10.1016/S0925-2312(98)00027-7
28. Trefethen LN, Bau D (1997) Numerical Linear Algebra. SIAM
29. Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2007) Numerical recipes 3rd edition: The art of scientific computing. Cambridge university press
30. Guennebaud G, Jacob B, et al (2010) Eigen v3. http://eigen.tuxfamily.org
31. Wilt N (2018) The CUDA Handbook: A Comprehensive Guide to GPU Programming, 2nd edn. ADDISON WESLEY Publishing Company Incorporated
32. Khronos OpenCL Working Group (2015) The OpenCL Specification. Version 2.0. Revision 29. Tech. rep.
33. NVIDIA Corporation (2015) CUDA occupancy calculator. https://developer.nvidia.com/
34. Volkov V (2015) Better performance at lower occupancy. In: Proceedings of the GPU Technology Conference, GTC, vol 10
35. Goossens B (2017) Quasar - optimization guide. https://quasar.ugent.be/files/doc/OptimizationGuide.html
36. Vinkler M, Havran V (2015) Register efficient dynamic memory allocator for gpus. In: Computer Graphics Forum, Wiley Online Library, vol 34, pp 143–154
37. Altera S (2014) for OpenCL best practices guide. Programming Reference, Altera (May 2015)
38. Takizawa H, Hirasawa S, Sugawara M, Gelado I, Kobayashi H, Hwu WmW (2015) Optimized data transfers based on the OpenCL event management mechanism. Scientific Programming 2015:2
39. Richter T, Pinheiro A, Schelkens P, Skodras A, Ebrahimi T (2016) Image Compression Grand Challenge at ICIP 2016. Tech. rep.