

# High-Performance Image Filters via Sparse Approximations

KERSTEN SCHUSTER, PHILIP TRETTNER, and LEIF KOBBELT,

Visual Computing Institute, RWTH Aachen University

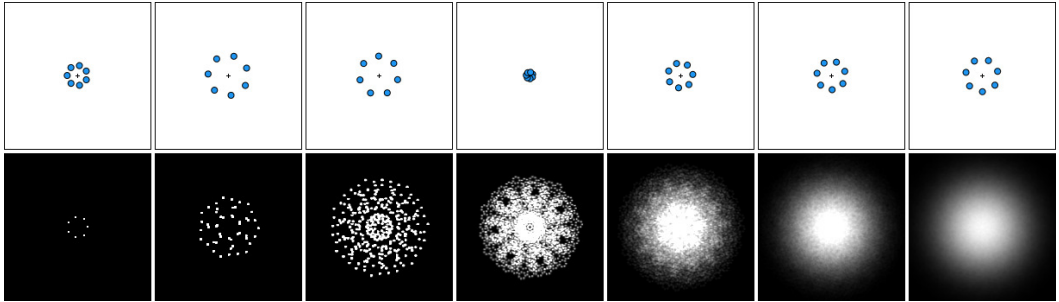


Fig. 1. Our method finds multi-pass sparse convolution filters that approximate a user-provided filter mask. For example, the shown filter consists of 7 passes each performing 7 image lookups with bilinear interpolation and is visually almost identical to a  $199 \times 199$  Gaussian blur. Top row shows per-pass sample positions, bottom row shows how the convolved passes converge towards the target mask. Our approximation requires only 49 samples, compared to 398 samples for a separated blur (200 samples with linear interpolation). While there is overhead for additional passes, our filter still only takes 0.74 ms while a separated Gaussian needs 1.64 ms with and 4.46 ms without interpolation and baking to shader code (on an NVidia GTX 1080 filtering a  $1920 \times 1080$  8-bit RGB image).

We present a numerical optimization method to find highly efficient (sparse) approximations for convolutional image filters. Using a modified parallel tempering approach, we solve a constrained optimization that maximizes approximation quality while strictly staying within a user-prescribed performance budget. The results are multi-pass filters where each pass computes a weighted sum of bilinearly interpolated sparse image samples, exploiting hardware acceleration on the GPU. We systematically decompose the target filter into a series of sparse convolutions, trying to find good trade-offs between approximation quality and performance. Since our sparse filters are linear and translation-invariant, they do not exhibit the aliasing and temporal coherence issues that often appear in filters working on image pyramids. We show several applications, ranging from simple Gaussian or box blurs to the emulation of sophisticated Bokeh effects with user-provided masks. Our filters achieve high performance as well as high quality, often providing significant speed-up at acceptable quality even for separable filters. The optimized filters can be baked into shaders and used as a drop-in replacement for filtering tasks in image processing or rendering pipelines.

CCS Concepts: • **Computing methodologies** → **Rendering**; *Image processing*.

Additional Key Words and Phrases: image filters, gaussian blurs, custom filter masks

Authors' address: Kersten Schuster, Schuster@cs.rwth-aachen.de; Philip Trettner, trettner@cs.rwth-aachen.de; Leif Kobbelt, kobbelt@cs.rwth-aachen.de,

Visual Computing Institute, RWTH Aachen University, Ahornstraße 55, Aachen, Germany, 52074.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, <https://doi.org/10.1145/3406182>.

**ACM Reference Format:**

Kersten Schuster, Philip Trettner, and Leif Kobbelt. 2020. High-Performance Image Filters via Sparse Approximations. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2, Article 14 (August 2020), 19 pages. <https://doi.org/10.1145/3406182>

**1 INTRODUCTION**

Filtering is an important tool in digital signal processing and is used for manipulating all kinds of data, e.g. for cleaning or separating certain parts. In the context of 2D and 3D imagery, filtering is frequently used to extract features like corners [Harris et al. 1988] or edges [Canny 1986], or entire frequency bands for the purpose of sharpening or blurring [Gonzales and Woods 2018]. Besides artistic usage, (Gaussian) blurs are an elementary component of rendering post-processes like *Bloom*, *Depth of Field* or *Screen-Space Ambient Occlusion* techniques. Especially large filters require a prohibitive number of texture fetches. Some filters, such as the Gaussian blur, are separable and thus can be decomposed into two passes of 1D blurs. On modern graphics hardware, bilinear interpolation of texture samples is especially efficient and can be exploited to roughly double the performance of a separated blur. Often, the exact filter shape is of secondary concern and approximations suffice. Kawase shows how a multi-pass filter with a simple pattern of four samples that lie exactly in the center of four pixels (and thus maximally exploit bilinear filtering) can be an effective approximation of a Gaussian blur [Kawase 2003].

In this work, we generalize this method and present an optimization framework based on a modified parallel tempering approach [Swendsen and Wang 1986] that decomposes almost arbitrary filter masks into a series of sparse convolutions. Each pass of the resulting filter computes a weighted sum of a small number of texture lookups that can lie at fractional positions to make use of bilinear filtering. We show how this strategy finds approximations that are simultaneously high-performance and high-quality. Even for the well-researched Gaussian blur, we find multi-pass filters that are significantly cheaper than the separated versions while taking only a minimal hit in quality. As our method approximates arbitrary target shapes, we demonstrate that it is possible to create inexpensive filter masks that simulate camera Bokeh effects or fulfill other artistic purposes.

In summary, we contribute:

- an optimization method for approximating image filters from user-provided masks via decomposition into multi-pass sparse convolutions
- new fast and high-quality approximations to Gaussian blurs, especially larger ones
- a collection of several ready-to-use filters for real-time rendering applications

In the remainder of the work we describe approaches related to our filtering methods in Section 2 and the algorithm details in Section 3. Results and evaluations of our optimization algorithms and produced filters are shown in Section 4. Finally, we show up limitations of the presented methods and possible areas for future research in Section 5 before concluding our work in Section 6.

**2 RELATED WORK**

The applications of image filters and even that of blurs are manifold as they range from 2D image optimization to machine learning and real-time rendering tasks. Due to the vast amount of conducted research in this field, we restrict ourselves to filtering methods that target interactive performance. A general overview over image filtering can be found in [Gonzales and Woods 2018].

**2.1 Gaussian and Box Filtering**

For graphics applications and especially rendering, discrete Gaussian blurs are typically truncated so that their support becomes finite and its performance increases. As the contribution of values is

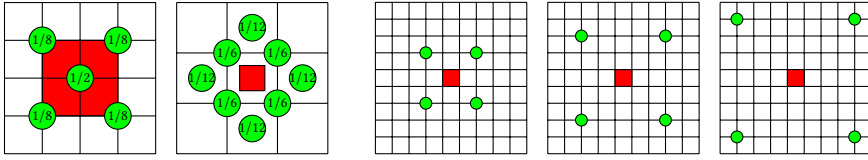


Fig. 2. Typical sampling stencils for the *Dual Filtering* (leftmost two images) and the *Kawase* method (rightmost three images). The *Dual Filtering* stencils have a fixed support in the downsampling and upsampling steps but perform sampling on various different resolutions. The *Kawase* passes (here a  $[1, 2, 3]$ -scheme) are typically run on the same resolution. The red center rectangles denote the currently processed pixel in the render target and the green dots show the sample positions within the input texture (sampling grid pictured in black). As the first two pictures describe downsampling and upsampling steps, the depiction of the processed pixel (red) is twice and half the sampling grid cell size, respectively.

rather low further away from the mean, the difference is usually negligible. Another justification for that is the decreased accuracy in rendering—typically 8 or 16 bit per value—that does not permit more than two or three digits of precision anyway. 2D Gaussian blurs are separable into two 1D blurs in orthogonal directions which decreases complexity from  $O(n^2)$  to  $O(n)$  without compromising quality. For larger filter sizes, however, the cost can still be prohibitively high, especially if the input image size is  $3840 \times 2160$  or beyond. The fact that small errors in a Gaussian filter mask are barely noticeable in the filtered result makes it attractive to trade accuracy for efficiency. The separated blur can be further improved by making use of bilinear sampling which is hardware-accelerated on modern GPUs [Rákóš 2010]. The actual sampling position has to be adapted to the individual weights of the two influencing pixels.

For box blurs, summed-area tables have been proposed to accelerate filtering in DirectX [Hensley et al. 2005] or CUDA [Nehab et al. 2011]. Another means to perform filtering on the GPU are Fast Fourier Transforms. The necessity of padding input images to power-of-two sizes, significant overhead for small kernel sizes and additional processing for multi-channel inputs [Fialka and Cadik 2006] makes them often impractical for use in rendering. An alternative usage for the (repeated) use of summed-area tables is demonstrated by Kosloff et al.. Rather than gathering pixel values, they introduce *filter spreading* which can be seen as reversing the gathering process [Kosloff et al. 2009] and enables the use of spatially-varying kernels. The currently fastest implementations of large low-order low-pass filters are based on a recursive filter formulation [Nehab and Maximo 2016; Nehab et al. 2011]. Their construction, however, relies on the separability of the filter kernel while we aim at arbitrary filter shapes.

A crude but fast approximation to a box blur is *mipmap filtering*. At first, a multiresolution pyramid is created from the input by successively averaging four neighboring pixels of the previous level. The final blur result is then picked as the mipmap level that most corresponds to the desired blur radius. Hardware-accelerated trilinear filtering enables sampling between neighboring pixels and consecutive detail levels. While the difference to a Gaussian blur is tremendous, the high performance can compensate for the poor quality in some situations.

The *Dual Filtering* approach by Bjørge performs multiple passes of downsampling and upsampling with constant filter masks [Bjørge 2015]. In contrast to naïve mipmapping, the stencils used for upsampling and downsampling produce a more circular shape which prevents blocky artifacts. Figure 2 (leftmost two images) shows exemplary stencils for the two different types of passes as proposed in [Bjørge 2015]. The advantage of this approach lies in the high performance which results from the lower number of overall processed fragments due to the intermediate downsampling. On the downside, dual filters have to be designed experimentally, as there is no generic translation

from blur radius to filtering levels or texel fetch offsets. Lee et al. propose an approach similar to mip mapping or the aforementioned *Dual Filtering* but perform non-linear interpolation between mip map levels [Lee et al. 2009b].

Multi-resolution methods are generally quite fast but the excessive downsampling of the input images can result in artifacts, especially in the vicinity of sharp features.

Kawase proposed a class of multipass blur algorithms that make use of hardware-accelerated bilinear filtering [Kawase 2003]. Every pass consists of four rotationally symmetric texture fetches and is thus simply defined by one of the 2D offsets to the processed pixel (cf. Figure 2, rightmost three images). To our knowledge, there is no generic way of approximating Gaussian filters of arbitrary size using it and filters are typically found experimentally. A typical choice resembling a  $35 \times 35$  Gaussian blur is e.g. a (0, 1, 2, 2, 3) offset tuple [Strugar 2014].

## 2.2 Non-Gaussian Filter Shapes

A typical real-time rendering use case for blurring is depth of field. Mimicking focal distance and aperture size of a real-world camera, the out-of-focus blur and other lens effects are simulated. Due to smoothing out towards the edges, Gaussian filtering cannot emulate the polygonal or circular shapes resulting from the diaphragm blades of cameras. When it comes to quality, ray tracing seems like a particularly suitable method to simulate e.g. bokeh effects of a given camera model. Early approaches include regular [Potmesil and Chakravarty 1982] and stochastic distribution of rays [Cook 1986] to simulate depth-of-field effects. More recently, Lee et al. proposed a combination of classical rasterization and additional lens-rays per pixel to account for out-of-focus effects and lens aberrations [Lee et al. 2010]. Joo et al. even consider lens imperfections due to manufacturing to focus on realism while still achieving interactive frame rates [Joo et al. 2016].

In rasterization, depth-of-field is either implemented as a scattering or a gathering technique. For scattering, every pixel is covered by a shaped sprite the size of the pixel's circle of confusion. All sprites are additively blended to account for typical bokeh effects. The sprite shape is dictated by a texture mask and can thus be arbitrary which makes this technique very versatile (cf. [Lee et al. 2008]). On the downside, larger circles of confusion—and thus larger sprites—create massive overdraw resulting in poor performance.

Gathering techniques, on the other hand, fetch weighted color samples within the vicinity of the evaluated pixel, typically within a certain shape area (cf. [Lee et al. 2009a]) and often in stochastic patterns. As circular or polygonal shapes are in general not separable, naïve implementations suffer either from low quality due to sparse sampling or from low performance due to exhaustive sampling.

Sousa separates the filtering of arbitrary shapes into a sparse gathering pass and a small-kernel blur that fills the gaps afterwards [Sousa 2013]. White and Barré-Brisebois describe the composition of hexagonal shapes by combining three rhombi in two linear-time passes [White and Barré-Brisebois 2011]. McIntosh et al. compose separable bokeh shapes from the intersection or union of parallelograms [McIntosh et al. 2012]. Besides being limited to a certain class of angular shapes, the used min/max operations are not linear and introduce artifacts if shapes overlap. Piponi generalizes the axis-aligned rectangles in box filters to convex polygons with vertices at integer positions [Piponi 2012]. Making use of summed-area tables, the number of needed texture fetches is independent of the kernel size. Moersch and Hamilton present a separation of circular and polygonal convolutions by writing into a three-dimensional render target between two consecutive passes [Moersch and Hamilton 2014]. Their method is limited to convex shapes and consumes a considerable amount of video memory. McGraw decomposes arbitrary filter shapes into a sum of multiple pairs of 1D convolutions using a singular value decomposition [McGraw 2015]. The resulting *low-rank linear*

*filters* consist of different pairs of horizontal and vertical passes. The user can provide the filter mask and choose the rank to affect the performance/quality trade-off.

[Gotsman](#) proposes a constant-time filtering approach for arbitrary and varying filter kernels, but requires a costly pre-filtering of the input image. [\[Gotsman 1994\]](#)

[Garcia](#) computes separable circular bokeh shapes in linear time facilitating convolutions in the frequency domain and complex number render targets [\[Garcia 2017\]](#).

We present an approach that aims for the best filter within a pre-defined budget. The method by [McGraw](#) [\[McGraw 2015\]](#) (see above) comes closest to that, but their overall number of texture fetches is at least twice the shape diameter and often multiple times higher (depending on the chosen rank). We facilitate a combination of multiple passes and hardware-accelerated bilinear filtering and show that this significantly increases performance especially for Gaussian blurs, but also for other filter shapes.

### 2.3 Texture Filtering

Interpreting textures as discrete samples of smooth images makes the need for filtered texture access apparent. Hardware-accelerated sampling is typically restricted to trilinear filtering (linearly interpolating between bilinear samples from two consecutive mip map levels), while higher-order methods are often desired for enhanced visual quality.

High-quality texture filtering methods have been proposed for isotropic [\[Manson and Schaefer 2013; Manson and Sloan 2016\]](#) and anisotropic sampling [\[Mavridis and Papaioannou 2011; McCormack et al. 1999\]](#). In these cases, the filtering kernels are typically very small and their current single-pass implementations are hard to beat. A multi-pass algorithm would most likely not increase performance, especially as additional passes add an overhead. Considering e.g. the case of performing bicubic filtering with only four bilinear samples [\[Sigg and Hadwiger 2005\]](#) (see [\[Djonov 2012\]](#) for implementation details), a two-pass algorithm that separates horizontal and vertical texture fetches, would still need four samples altogether. A similar situation is given in [\[Manson and Sloan 2016\]](#) in the context of pre-filtering environment cube maps in real-time: Bilinear sampling is used to reduce the number of texture fetches from 16 to 4 for quadratic B-spline filtering in mip map generation [\[Manson and Sloan 2016\]](#).

## 3 METHOD

Let  $A$  be the convolution kernel for an image filter that we want to optimize. Given an input image  $f(x, y)$ , the output is computed via

$$g(x, y) = A * f(x, y). \quad (1)$$

Typical examples for  $A$  are Gaussian blurs, box filters, Bokeh shapes for depth-of-field, or Airy disks for bloom.

While image filters are ubiquitous in real-time applications, they often present significant challenges to evaluate within the computational budget. Our goal is to find alternative filters  $B$  that are substantially cheaper to evaluate and approximate  $A$  sufficiently well. Especially large blurs tend to be expensive even though their result is typically very smooth and small imperfections in the filter are hardly visible.

### 3.1 Sparse GPU Filters

Separable filters and the Kawase blur have in common that they decompose the large filter  $A$  into multiple filters that, when applied sequentially, result in the original filter, or an approximation thereof. We generalize this and formulate the following task:

Given a convolutional image filter  $A$ , we solve the following *filter approximation problem with constrained budget*:

$$\begin{aligned} \min_{B_1, \dots, B_n} \quad & L(A, B_1 * \dots * B_n) \\ \text{s.t.} \quad & \text{cost}(\{B_1, \dots, B_n\}) \leq C \end{aligned} \quad (2)$$

The result is a filter  $B = B_1 * \dots * B_n$  that approximates  $A$  and stays within the budget limit  $C$ . The loss function  $L(\cdot, \cdot)$  can be freely chosen. We motivate three choices for different use cases in Section 3.3.

$B$  is intended to be implemented as a multi-pass GPU algorithm with  $n$  passes, either in fragment or compute shaders. Each  $B_i$  computes a weighted sum of  $k_i$  samples:

$$(B_i * f)(x, y) = \sum_{j=1}^{k_i} f(x + \Delta x_j, y + \Delta y_j) \cdot w_j \quad (3)$$

The sample offsets in  $f(x + \Delta x_j, y + \Delta y_j)$  can be fractional, resulting in a bilinear interpolation of the four neighboring integer locations. This interpolation is especially efficient on the GPU and the main motivation behind the Kawase blur because a single texture lookup effectively fetches four samples at once.

$\text{cost}(\cdot)$  is an estimate of the computational costs of applying the filter  $B$ . The actual costs are extremely hard to quantify as they depend on many GPU internals, such as texture cache behavior, on-the-fly texture compression, and shader scheduling. We found the total number of samples plus per-pass fixed costs to be a suitable measure in practice:

$$\text{cost}(\{B_1, \dots, B_n\}) = \lambda \cdot n + \sum_{i=1}^n k_i \quad (4)$$

where  $\lambda$  is a user parameter.

Figure 1 shows the structure and result of a  $n = 7$ ,  $k_i = 7$  filter fitted to a  $199 \times 199$  Gaussian blur.

### 3.2 Optimization

The optimization problem posed in the previous section is challenging to solve. We have discrete variables  $n$  (number of passes) and  $k_i$  (number of samples per pass) as well as continuous variables,  $\Delta x_j$  and  $\Delta y_j$  (sample offsets), and  $w_j$  (sample weight). The number of variables changes if the structure of the filter (number of passes or samples) changes. Any change in any variable has almost global influence since changing even a single sample often affects more than 50% of  $B$ 's impulse response.

Due to the mixture of continuous and discrete variables and their indirect, non-local influence, purely gradient-based approaches perform poorly on our problem. Even if only used on the continuous variables, gradient descent or second order methods tend to converge to bad local minima.

We achieved the best results with a modified *parallel tempering* [Swendsen and Wang 1986] approach. A set of  $k$  candidate filters  $B^i$ , each consisting of multiple passes  $\{B_1^i, \dots, B_n^i\}$ , is maintained. In each iteration, the filters are randomly mutated with decreasing magnitude, from a big change of  $B^1$  to a small change of  $B^k$ . These are called *temperature bands*. The changed filter  $B'^i$  replaces the previous candidate  $B^i$  with a probability of

$$p = \min \left( 1, \frac{L(A, B^i)}{L(A, B'^i)} \right), \quad (5)$$



also known as the *Metropolis-Hastings criterion*. If the new filter is better, it is always accepted, otherwise only with a probability based on the difference in approximation quality. After a certain number of iterations, all  $B^i$  are replaced by the current best filter (the *synchronization* step).

Filters with more passes or samples can achieve better approximation quality but are also more complex to optimize. Accordingly, the optimization tends to get stuck in local minima of too complex filters and the probability of jumping into a better configuration is low. We alleviate this by employing a *niche* technique as is often used in evolutionary optimization: Instead of running a single *parallel tempering*, we concurrently run  $c$  instances where each instance  $i = 1..c$  is responsible for the filter cost bracket  $(\frac{i-1}{c} \cdot C, \frac{i}{c} \cdot C]$ . When randomly changing a filter causes a change in costs, it is moved to the appropriate instance. In this way, cheaper but lower-quality filters can be used as stepping stones to find good local minima and evolve into more costly, higher-quality ones. We achieved the best results with 5–10 temperature bands and 3–6 cost brackets, depending on the target filter complexity (lower complexity requires less bands and brackets).

**3.2.1 Candidate Mutation.** Given a candidate  $B$  in a temperature band  $T \in 1..k$ , we perform three different types of changes when creating a new candidate  $B'$ .

First, we apply *continuous* mutations by adding random values up to  $2^{-T}$  to each sample offset  $\Delta x_j$  and  $\Delta y_j$ . The sample weights  $w_j$  are more sensitive to change, so we only add up to  $2^{-T}/10$  to them.

This will never alter the structure or cost of a filter, so we also apply *morphological* mutations that modify the filter structure (number of passes or samples) while trying to maintain the same loss value. These operations do not always seem useful in isolation but in combination with the continuous mutations they allow efficient sampling of the variable-dimensional optimization space, similar to how *reversible-jump MCMC* [Green 1995] works. We found the following operations to be useful:

- *merge samples*: replace two samples of the same pass by a single one, adding their weights and randomly interpolating between their sample offsets.
- *split samples*: duplicate a sample, randomly splitting the previous weight.
- *remove sample*: randomly remove a sample from a pass.
- *add zero sample*: add a sample of zero weight to a pass.
- *merge passes*: replace two passes  $a$  and  $b$  by an approximate convolution of them: for each sample pair  $(s_a, s_b) \in a \times b$ , create a new sample with sample offsets added and weights multiplied. To prevent an explosion of sample counts, we immediately *merge samples* until a maximum sample count is met.
- *remove pass*: randomly remove a pass from the filter.
- *add neutral pass*: add a new pass consisting of samples with random offset and zero weight and samples with zero offset and weights that sum to 1.

Finally, we employ *domain-specific* mutations that try to create patterns that we observed in good filters:

- *normalize weights*: ensure that the sample weights sum to 1 for each pass.
- *radial symmetry*: Take a random sample from a pass  $B_i$  to compute radius  $r$  and angle offset  $\alpha$ . All sample offsets  $\Delta x_j$  and  $\Delta y_j$  of that pass are then set to  $r \sin(2\pi j/k_i + \alpha)$  and  $r \cos(2\pi j/k_i + \alpha)$  with weight  $1/k_i$ , creating a circular pattern. Note that this contains the Kawase blur pattern as a special case ( $k_i = 4$ ,  $\alpha = \pi/4$ ).

Not all operators are applied for every candidate. Each operation has a certain probability to be applied. We use between 1–5% per operation for the topological operations, scaling linearly with  $T$ . Radial symmetry is enforced with 5% probability on a single, randomly chosen pass and

with another 5% on the whole filter. If weight normalization is employed, we apply it with a 90% probability. Figure 4 shows how the domain-specific operations speed up the optimization convergence.

### 3.3 Loss Function

Purely convolutional image operations are *linear* and *translation-invariant* (shifting the input in any direction shifts the output by the same vector). Such *LTI* filters are fully defined by their impulse response  $(f * \delta)(x, y)$ , i.e. by their output when given a black input with a single white pixel in the center. Thus, a loss function  $d(A, B)$  that is based on the impulse response difference is independent of concrete images.

We found that there is no universal loss function and the choice depends on the intended use case. However, a few building blocks are useful. Note that common perceptually-motivated image metrics cannot be used as impulse responses are not directly perceived.

The root-mean-square error (RMSE) of the impulse response difference is a good base indicator of the approximation quality:

$$L_{\text{RMSE}}(A, B) = \sqrt{\frac{1}{N} \sum_{x,y} \|(A * \delta)(x, y) - (B * \delta)(x, y)\|^2} \quad (6)$$

where  $N$  is the number of non-zero pixels in the bounding box of  $(A * \delta)(x, y)$ . However, the sign of the difference at individual pixels is not considered. Thus, an average 5% deviation might conserve the image brightness (if the deviations are uncorrelated), or make the image 5% darker or 5% brighter (if strongly correlated). This is often undesirable but can be prevented by penalizing differences in filter energy:

$$L_{\text{ENERGY}}(A, B) = \left| \sum_{x,y} (A * \delta)(x, y) - \sum_{x,y} (B * \delta)(x, y) \right| \quad (7)$$

**3.3.1 Blurs.** Filters that are intended to be used as blurs benefit from the error-distributing tendency of RMSE. However, energy preservation is quite important as image-wide brightness changes are immediately visible:

$$L_{\text{BLUR}}(A, B) = L_{\text{RMSE}}(A, B) + M \cdot \max(0, L_{\text{ENERGY}}(A, B) - \tau), \quad (8)$$

where the second part is used as a soft barrier with a large factor  $M = 100$  penalizing the amount of  $L_{\text{ENERGY}}(A, B)$  that surpasses a user-defined threshold  $\tau$ . We use  $\tau = 0.01$  to avoid blurs that change brightness by more than 1%.

**3.3.2 Shape-Preservation.** For filters where the shape is very important, e.g. for Bokeh or bloom effects, we found filters of subjectively better quality by breaking symmetry: It is more noticeable if  $B$ 's impulse response has a bright pixel where  $A$ 's is dark than the other way around. We model this by defining the weighted impulse response difference

$$D_{\delta}(x, y) = [(A * \delta)(x, y) - (B * \delta)(x, y)] \cdot \left( 1 + \omega \cdot \left( 1 - \frac{(A * \delta)(x, y)}{\max_{x',y'} (A * \delta)(x', y')} \right) \right), \quad (9)$$

where  $\omega$  is a user parameter which we e.g. set as  $\omega = 3$  for the approximation of custom filter shapes in Section 4.4.

Intuitively, in  $D_{\delta}(x, y)$ , differences are weighted by 1 for regions that are “bright” in  $A$  and by  $1 + \omega$  for regions that are “dark”. This can be used in a modified RMSE loss:



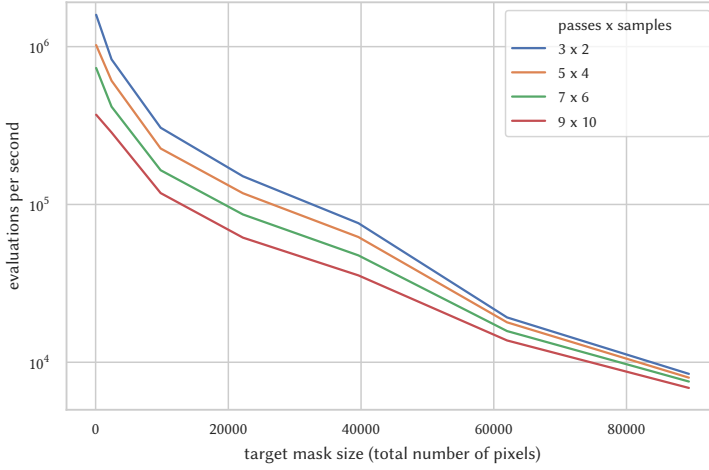


Fig. 3. Candidate evaluations per second depending on target mask size. For small targets and low numbers of passes and samples, one million candidates can be evaluated per second. For larger masks, the throughput decreases drastically. Evaluated masks are square and have side lengths between 9 and 299 pixels.

$$L_{\text{SHAPE}}(A, B) = \sqrt{\frac{1}{N} \sum_{x,y} \|D_{\delta}(x, y)\|^2} \quad (10)$$

**3.3.3 Overfitting.** The previously described losses are defined on the impulse responses. While we motivate different losses for different use cases, these proxy metrics are not perfect. If the use case is clearly defined, e.g. a specific effect used in a game with a distinct visual style, it is also possible to directly “overfit” the filter on a set of representative images  $I \in \mathcal{I}$ :

$$L_{\text{OVERFIT}}(A, B) = \sum_{I \in \mathcal{I}} d'(A * I, B * I), \quad (11)$$

where  $d'(\cdot, \cdot)$  is a user-provided metric on *images*, e.g. PSNR or SSIM [Wang et al. 2004]. While this loss is less general and considerably more expensive to evaluate, it can produce superior results if applicable. Evaluation of loss functions and examples of fitted filters can be found in Section 4.

### 3.4 Candidate Evaluation

The runtime of our optimization is dominated by the evaluation of the loss function. All the losses we describe in Section 3.3 require applying the candidate filter on either an impulse or a reference image. While our filters are by design quite fast, we still need to evaluate millions of candidates during optimization. To make the evaluation fast, we batch together 200–1000 candidates, upload their parameters into a GPU buffer, and use compute shaders to evaluate them. Candidate filters are padded with trivial passes up to the maximum number of passes in the batch. Two 2D array textures are used in a ping-pong fashion to compute the filter response with one shader dispatch per pass. A final compute shader evaluates the selected loss function. While this generic data-driven filter evaluation is not as fast as the shaders that we generate for a given, fixed filter, we can nevertheless evaluate up to a million candidates per second on a modern GPU. For the  $199 \times 199$  Gaussian

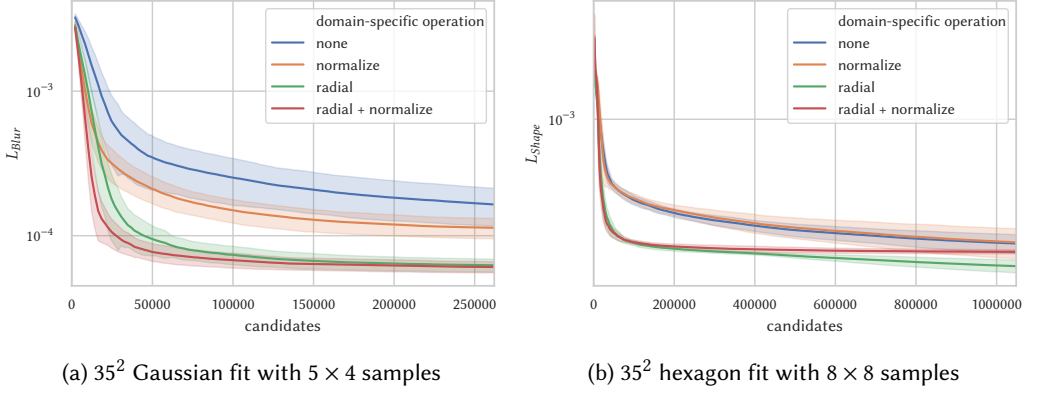


Fig. 4. Convergence behavior of two different scenarios, Gaussian via  $L_{BLUR}$  (easy) and hexagon via  $L_{SHAPE}$  (challenging). Average and standard deviation over 50 runs are shown. Our domain-specific candidate mutations speed up convergence if they synergize with the scenario.

example from Figure 1, one million candidates with 7 passes and 7 samples took approximately 21 s to evaluate on an NVidia GTX 1080. For a  $49 \times 49$  input, only 1.64 s were needed for the evaluation of the same amount of  $5 \times 4$ -filters. Figure 3 describes the evaluation throughput depending on target mask size. Using the generated filters during optimization is not an option as compiling individual shaders per candidate is prohibitively expensive, even if the actual evaluation is faster.

## 4 EVALUATION

### 4.1 Optimization

Fig. 4 demonstrates the convergence behavior of our optimization and justifies our domain-specific operations introduced in Section 3.2. In most scenarios, the optimization quickly finds a good first approximation and then slowly fine-tunes the filter.

Our method works best on Gaussian blurs using  $L_{BLUR}$  where it converges fast and finds excellent solutions. Here, both domain-specific operations are beneficial. As the loss penalizes non-normalized results, frequent re-normalization produces more promising candidates. The larger improvement is gained by occasionally restoring radial symmetry in some passes. These passes tend to work well for many symmetric shapes and reduce the effective search space significantly, thus the improved convergence speed. Note that strict radial symmetry is not optimal due to the bilinear filtering and the discretized pixel grid. Thus, these operations are only applied probabilistically and the per-parameter continuous mutation finds a local minimum in the vicinity of the symmetric filter.

Fitting the hexagon using the  $L_{SHAPE}$  loss is more challenging. Filter normalization is not important for the loss and seems to hinder exploration at the tail end of the optimization. Exploiting radial symmetry still speeds up convergence considerably which makes sense due to the symmetric target shape. For more complex targets we often see significant loss improvements at unpredictable points far into the optimization. This manifests itself as a widening standard deviation on the long tail and seems to indicate that the optimization jumps into regions with better local minima. For even more complex cases (larger filter masks, non-convex shapes, more passes and samples per pass) we have seen significant progress well after 10 million candidate evaluations.

The main difficulty of the optimization process can be seen in Fig. 5. Slicing through the loss landscape makes it immediately visible that the optimization has to work around a myriad of local minima. Especially during the beginning many clusters of filter configurations lead to plausible

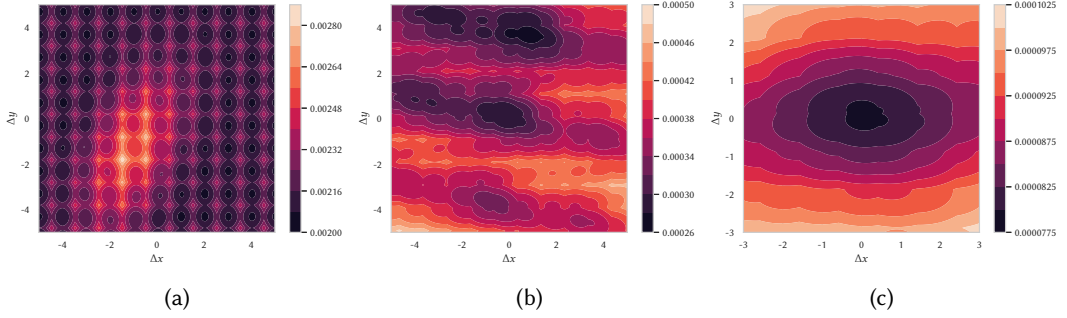


Fig. 5. Plotting the loss landscape reveals the optimization challenges of our problem. The images show the loss when a single filter sample is moved: (a) is taken during fitting a hexagon using  $L_{\text{SHAPE}}$ , (b) for a Gaussian using  $L_{\text{BLUR}}$ , both after  $10^4$  candidates. This highly non-convex loss landscape prevents the use of simple gradient-based optimization. Only close to convergence does the problem become locally convex, e.g. in (c) for fitting a dollar sign using  $L_{\text{RMSE}}$  after  $10^7$  candidates. However, this is usually only a local minimum in our high-dimensional optimization space.

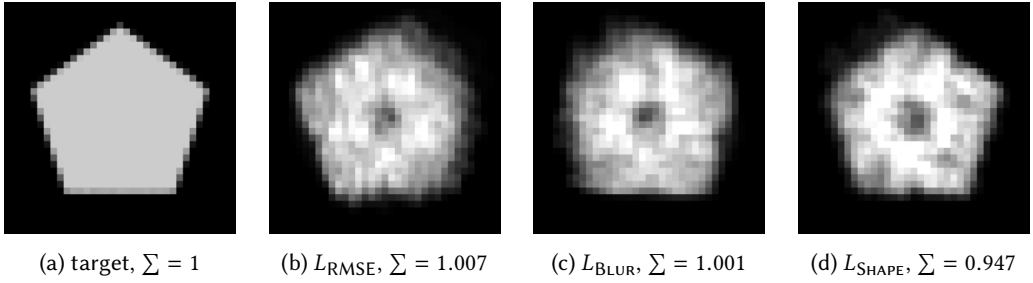


Fig. 6. The difference between our proposed loss functions becomes more pronounced when working with a limited performance budget. Shown are the best 3-pass 9-samples-per-pass filters found after 200 million candidate evaluations. The different losses lead to different trade-offs:  $L_{\text{BLUR}}$  is an energy-preserving version of  $L_{\text{RMSE}}$  and  $L_{\text{SHAPE}}$  ( $\omega = 3$ ) produces crisper shape boundaries at the cost of a higher RMSE.

low-loss regions in our search space. While the problem looks locally convex closer to convergence, these are only local minima. The parallel tempering always has a certain probability to jump to other minima, which is what happens in the tail of Fig. 4 (b). However, larger cases have well above 100 continuous dimensions with many symmetries, which makes finding the global optimum progressively unlikely. An interesting test is fitting a Dirac impulse with an unnecessarily large number of passes and samples. Due to the fact that only one pixel is non-zero in the input image (and also in the filter response), there are countless combinations of passes and samples that minimize the energy. Applying that filter to an image, however, leads to an unwanted result because of several (possibly negative) non-centered samples. This situation is a rather unusual one but could be dealt with e.g. by reducing the maximum number of passes to a reasonable amount or by using the  $L_{\text{OVERFIT}}$  loss function.

## 4.2 Loss Function

The loss functions from Section 3.3 correspond to different use cases. An example of their difference is depicted in Fig. 6. In general,  $L_{\text{RMSE}}$  will not produce energy-conserving filters. This becomes

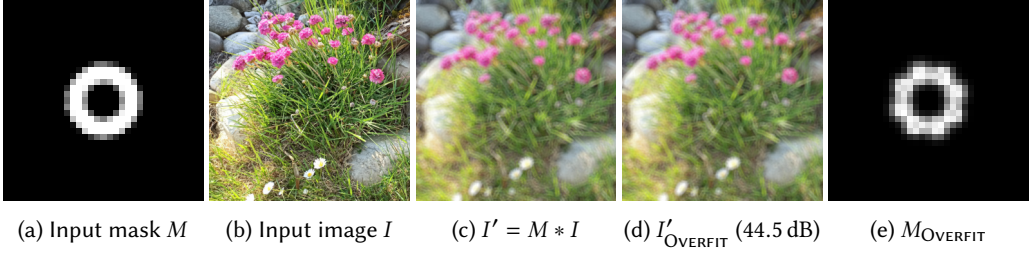


Fig. 7. The  $L_{\text{OVERFIT}}$  loss function can be used to estimate the applied filter  $M$  (a) given an unfiltered image  $I$  (b) and the filtered result  $I'$  (c). The optimized filter was constrained to 3 passes with 8 samples each. Its impulse response is depicted in (e) and the convolution result of  $I * M_{\text{OVERFIT}}$  is shown in (d). Optimizing only the impulse response with  $L_{\text{BLUR}}$  leads to a slightly worse result in this example (43.89 dB, not shown here).

especially pronounced with a limited budget where larger trade-offs are made. The example shows a fit that will produce 2.7% more energy, which results in a noticeable change in image brightness. Instead of re-normalizing the filter after fitting, we model energy-preservation directly in  $L_{\text{BLUR}}$ . When used as Bokeh (e.g. Fig. 10, column “Detail”), the overall shape is more important than the RMSE, which is what  $L_{\text{SHAPE}}$  is designed for.

Another approach of finding an approximate filter is to use the  $L_{\text{OVERFIT}}$  loss function that “overfits” a filter to produce an image  $I'$  from another image  $I$  (cf. Figure 7). The drawback of this method is that it has to convolve the input image  $I$  for every loss evaluation which is much more expensive than computing the impulse response of the evaluated filter.

It is important to note that loss functions are only evaluated within the texture area uploaded to the GPU but the optimized filters can sample outside of that area. While boundary handling can be specified explicitly, the optimizer can take advantage of that and sample out-of-bounds deliberately to reduce costs. We never experienced that behavior for Gaussian filter kernels, but for more complex shapes like those in Figure 9. A simple yet effective way of circumventing this problem is to increase the padding of the input mask at the cost of optimization speed.

### 4.3 Filter Comparison

In Table 2 and Figure 8 we compare the presented strategies in terms of performance and quality for different input image resolutions. The evaluated filtering technique is a  $97 \times 97$  truncated  $3\sigma$ -Gaussian blur. As ground truth we compute the weights of a 2D Gaussian directly in the shader because even the separation introduces rounding errors and floating point inaccuracies. The best approximation to that is a separated Gaussian. For further performance improvement we make use of hardware-accelerated linear interpolation and additionally bake filtering weights as constants into shader code. In general, the timings contain all costs that occur every frame for dynamic scenes, e.g. the mipmap creation for the mipmap filtering technique. The *Dual Filtering* consists of 4 downsampling and 4 upsampling passes while the *Kawase blur* consists of 9 passes with 4 samples each. Finally, our approximation is a 5-pass shader algorithm with 5 texture fetches per pass. It is worthwhile noting that for small kernels (below  $35 \times 35$ ) a well-tweaked separated Gaussian is a reasonable choice. For larger kernels, however, the costs do not scale very well, especially for fullscreen blurs. This is also depicted in Table 1 where we show approximations for Gaussian filters of different kernel sizes.

We found that even if the approximation quality is so low that artifacts in the impulse response are clearly visible (cf. Table 1, top row), the filtered results are still very close to the ground truth which might be preferable to a solution that is more exact but also more expensive. Due to decreased

Table 1. Approximations for Gaussian Filters of different kernel sizes. Quality and cost increase from top to bottom. Kernel size increases from left to right. Overlay consists of passes  $\times$  samples-per-pass and filtering time for a  $1920 \times 1080$  RGB 8-bit target. Ground Truth Gauss filtering (“G.T.”, marked with an asterisk, bottom row) is separated, makes use of hardware-accelerated linear interpolation and was baked into shaders for maximum performance.

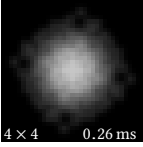
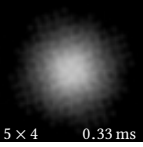
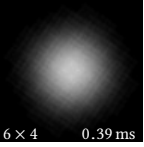
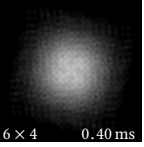
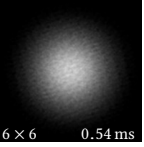
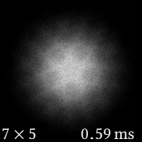
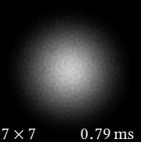
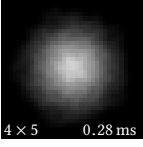
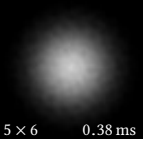
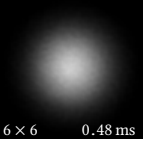
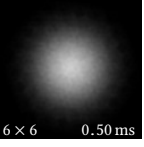
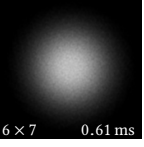
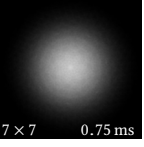
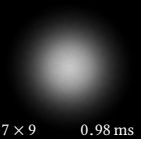
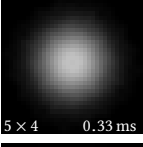
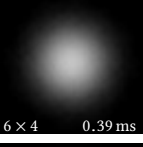
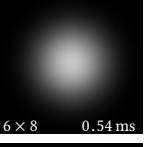
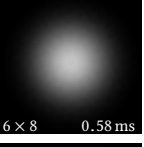
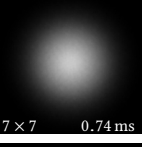
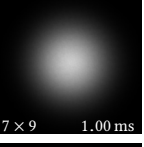
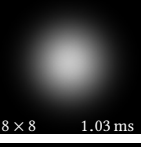
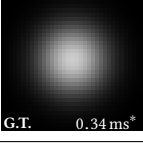
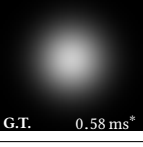
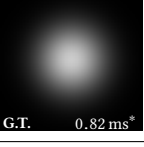
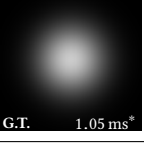
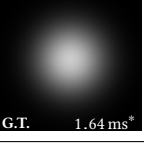
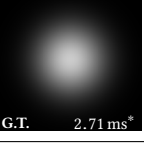
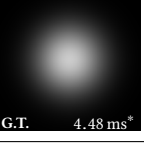
35 $\times$ 35	65 $\times$ 65	95 $\times$ 95	125 $\times$ 125	199 $\times$ 199	333 $\times$ 333	555 $\times$ 555
 4 $\times$ 4 0.26 ms	 5 $\times$ 4 0.33 ms	 6 $\times$ 4 0.39 ms	 6 $\times$ 4 0.40 ms	 6 $\times$ 6 0.54 ms	 7 $\times$ 5 0.59 ms	 7 $\times$ 7 0.79 ms
 4 $\times$ 5 0.28 ms	 5 $\times$ 6 0.38 ms	 6 $\times$ 6 0.48 ms	 6 $\times$ 6 0.50 ms	 6 $\times$ 7 0.61 ms	 7 $\times$ 7 0.75 ms	 7 $\times$ 9 0.98 ms
 5 $\times$ 4 0.33 ms	 6 $\times$ 4 0.39 ms	 6 $\times$ 8 0.54 ms	 6 $\times$ 8 0.58 ms	 7 $\times$ 7 0.74 ms	 7 $\times$ 9 1.00 ms	 8 $\times$ 8 1.03 ms
 G.T. 0.34 ms*	 G.T. 0.58 ms*	 G.T. 0.82 ms*	 G.T. 1.05 ms*	 G.T. 1.64 ms*	 G.T. 2.71 ms*	 G.T. 4.48 ms*

Table 2. Comparison of different filtering algorithms in terms of performance and quality for a  $97 \times 97$  truncated  $3\sigma$ -Gaussian blur on an Nvidia GTX 1080. Algorithms were applied on the photo shown in Figure 8. Time is in ms and peak signal-to-noise ratio (PSNR) in dB. Render targets have 16 bit resolution per channel. Apart from the two-dimensional Gaussian (Ground Truth), all weights are baked into shaders for increased performance. The number of samples for the separated Gaussian are halved by making use of linear interpolation.

Size	2D Gauss		Sep. Gauss		Mipmap		Dual Filter		Kawase		Ours 5x5	
	Time	PSNR	Time	PSNR	Time	PSNR	Time	PSNR	Time	PSNR	Time	PSNR
2160p	347.74	$\infty$	3.27	74.91	0.49	29.99	0.77	41.02	2.89	40.62	2.04	49.98
1080p	88.39	$\infty$	0.84	74.20	0.14	26.43	0.22	37.82	0.79	37.51	0.54	47.38
720p	39.73	$\infty$	0.39	74.04	0.07	24.81	0.11	36.24	0.39	35.54	0.26	45.85
540p	22.80	$\infty$	0.23	74.05	0.05	23.80	0.08	34.67	0.25	34.32	0.16	44.65

intermediate render target sizes, *Dual Filtering* is faster than the presented approach, but also less accurate. Furthermore we found intermediate downsampling followed by upsampling to result in visible quantization artifacts for moving objects. When images are sampled out-of-bounds we typically mirror the coordinates at the boundaries. We found that clamping to boundary values or setting those samples to zero leads to noticeable intensity inconsistencies. However, boundary handling is supported by hardware and is independent from the actual filter shaders. Note that for the loss evaluation in the optimization procedure, we generously pad with zeros as mirroring would lead to artifacts (cf. 4.2).

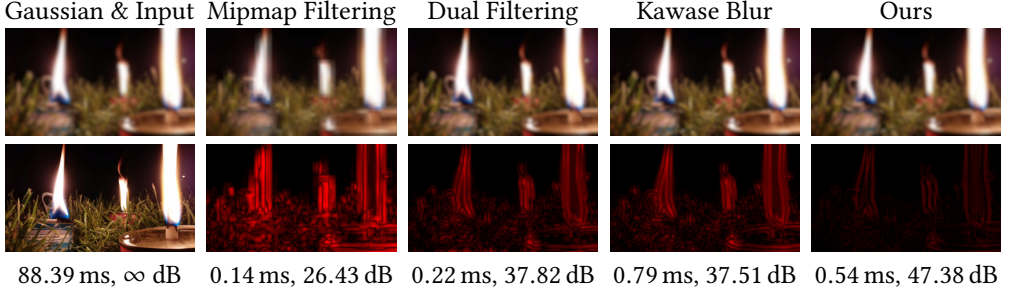


Fig. 8. Comparison of different filtering techniques in terms of quality for a  $97 \times 97$  truncated  $3\sigma$ -Gaussian blur. Top row shows filtered results. Bottom row shows input image and differences between ground truth and respective result. Color varies from 0% (black) to 20% (red) color difference. Measure is Euclidean differences with image colors interpreted as 3D vectors. Input image size is  $1920 \times 1080$  and render targets have 16 bit resolution per channel.

#### 4.4 Custom Filter Masks

Figure 9 shows examples for several custom filter masks.

In Figure 10 we evaluate performance and quality of a circular and a hexagonal filter approximated by our method. The target is a high dynamic range photo (16 bit per color channel) with several small bright lights. Filtering it with one of the chosen masks creates an out-of-focus effect. Our circle approximation takes about one millisecond on an NVidia GTX 1080 and is thus almost three times faster than the method of McGraw [McGraw 2015], which is roughly comparable in quality. Our filter consists of four passes with 32, 31, 27 and 7 samples, i.e. 97 samples altogether. The low-rank approximation consists of three separable filters ( $3 \cdot 2 \cdot 61$  fetches) that add up (+3 fetches) to the final result (369 fetches). We chose  $k = 3$  for the number of ranks (and thus three separable filters) as a lower amount leads to a drastic decrease in quality and a higher amount would increase evaluation costs even further. Our hexagonal filter consists of four passes with 32, 32, 27, and 12 samples. The low-rank counterpart performs  $53 + 61$  texture fetches for each of its three ranks (the height of the hexagon is less than its width) and additional three fetches for computing the sum, i.e. 345 fetches in total. In contrast, our hexagon is computed with 103 samples and two passes less.

Figure 11 gives an example for the distribution of individual texture sample positions for a custom mask. The ground truth is a rasterized version of the character string “2020” with 3150 non-zero pixels. The filter was optimized using the  $L_{\text{SHAPE}}$  loss ( $\omega = 3$ ) and the five individual passes consist of 12, 27, 32, 14 and 14 samples.

## 5 LIMITATIONS AND FUTURE WORK

Regarding Gaussian filtering, our approximations are best suited for medium-sized filters. Our experiments show that our approximations cannot beat efficiently implemented separated Gaussians with kernels smaller than approx.  $35 \times 35$ . For very large kernels, the more involved implementation of recursive Gaussian filters [Nehab and Maximo 2016; Nehab et al. 2011] might be worth the effort.

Our arbitrary filter masks often lack consistency in brightness which is clearly visible in the impulse responses, but usually less prominent in the filtered results (cf. Figure 9).

We currently only consider filters of the form  $B_1 * \dots * B_n$  where each  $B_i$  is a sparse convolution. There are different patterns that also result in linear, translation-invariant filters. We believe that integrating additive components and, for example, allow filters of the form  $(B_1 * B_2 + B_3 * B_4) * B_5 + B_6 * B_7$ , is promising. Especially for complex shapes this would allow a “regional decomposition”



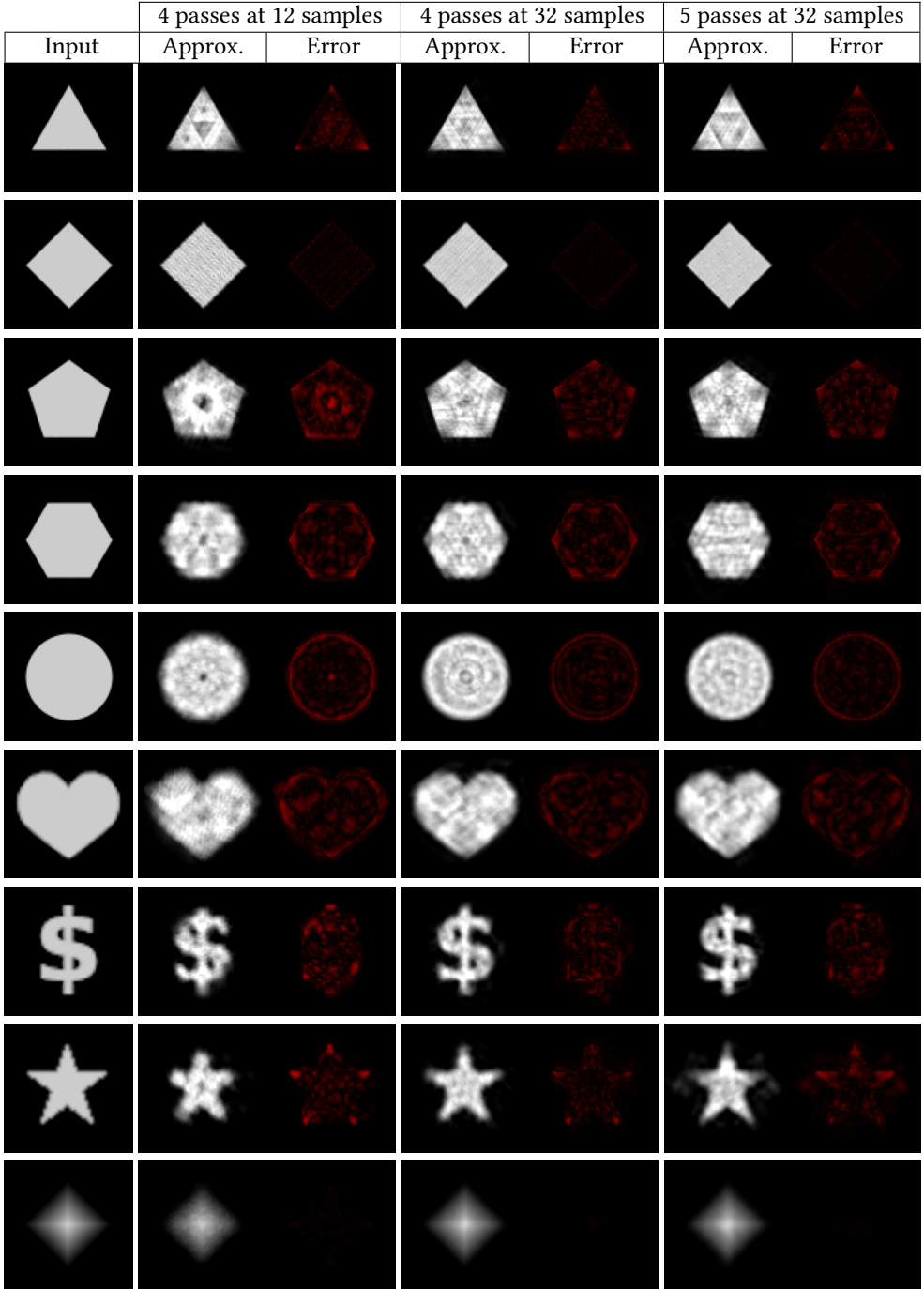


Fig. 9. Examples for approximations of custom filter masks produced by our method. Error images show absolute difference between ground truths and approximations. Masks have between 1200 and 3400 non-zero pixels and the used loss function was  $L_{\text{SHAPE}}$  ( $\omega = 3$ ).

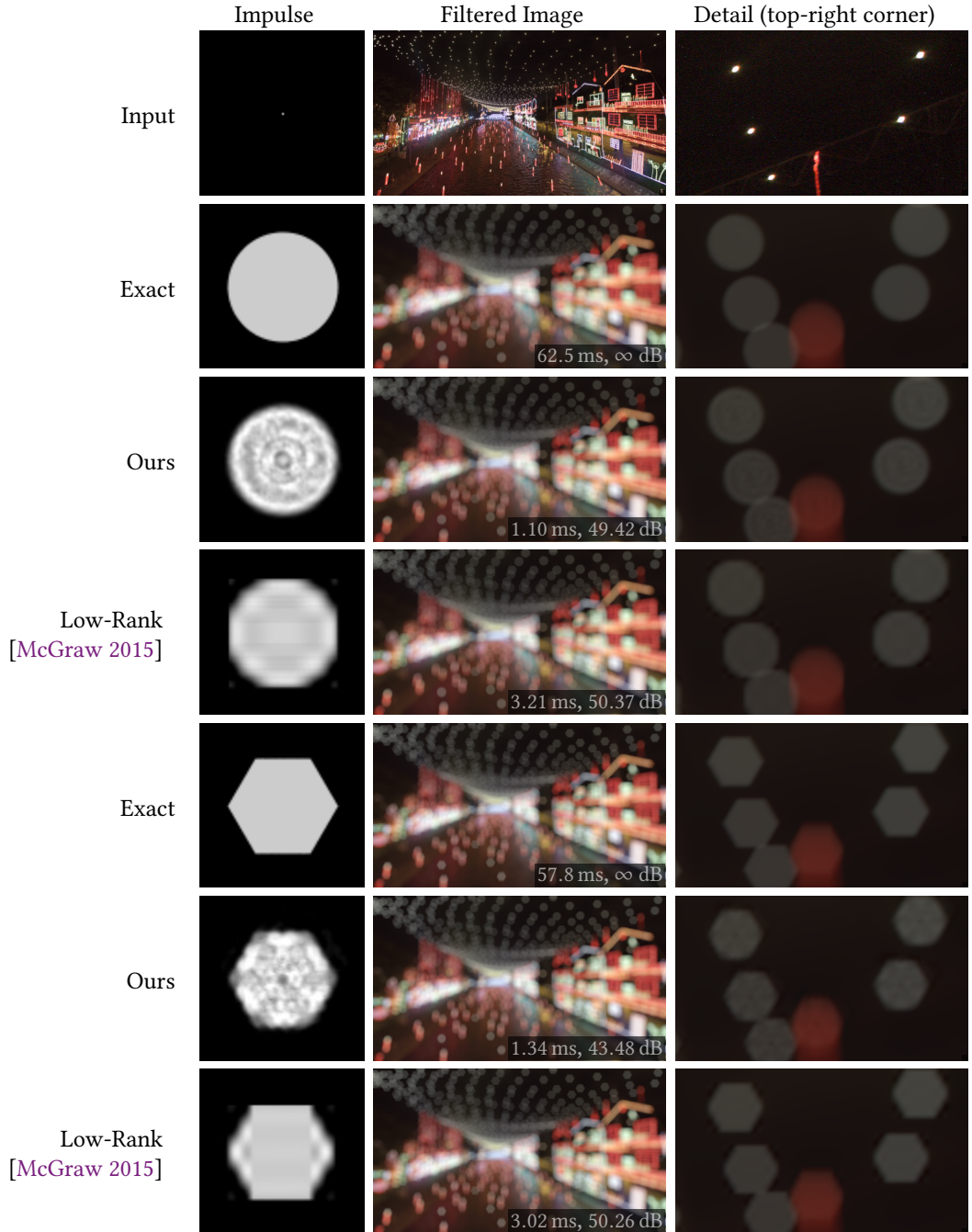


Fig. 10. Simulating a circular and a hexagonal out-of-focus effect on a 1080p high dynamic range image. Filtering was performed directly on the 16 bit input channels. Peak signal-to-noise ratios have been computed on the final 8 bit images after tone mapping (which are shown here). See Section 4.4 for details. Input image courtesy Sylvia Schuster.



Fig. 11. Example for a 5-pass filter resembling the characters “2020”. Top row shows the individual sample positions (blue) in each pass while the bottom row shows the filter response after convolving the individual passes (negative weights displayed in orange). Applying the filter to a  $1920 \times 1080$  RGB target takes 1.028 ms for 8 bit precision and 1.574 ms for 16 bit precision.

into parts that can be efficiently convolved. This would also subsume the low-rank approximations of [McGraw 2015], which produce filters of the form  $\sum_i B_x^i * B_y^i$ . The main challenge is to guide the optimization into good minima.

While having an *LTI* filter is extremely beneficial for consistent quality and temporal coherence, in practice many post-processing effects operate on down-sampled targets to achieve acceptable performance. It should be straightforward to integrate down-sampling and up-sampling passes into our filters, though open questions are how to compute the loss (there is no unique impulse response anymore) and if the samples should depend on the subpixel offset.

The critical aspect for finding our sparse filters is the optimization procedure. While it already reliably finds filters of high quality and performance, there is still room for improvement. There are many hyperparameters and while most of them mainly affect convergence speed, it would be preferable if they could be auto-configured or even self-adapting. For large filters we currently rely on the domain-specific operations as they drastically speed up convergence. A promising avenue for future research would be to extend the set of domain-specific operations from Section 3.2.1 or even automatically discover good filter patterns that, when added as another operation, result in faster convergence, similar to what can be seen in Fig. 4.

In Table 1 we have given Gaussian filter examples for different performance/quality trade-offs. However, those were only benchmarked on desktop GPUs. For mobile GPUs, it would be interesting to take tiled rendering into account when evaluating approximated filters.

Finally, another interesting avenue is to create dynamic filters that can change their shape based on per-pixel parameters. For example, a Gaussian blur where the blur size can be changed per-pixel. This could be achieved by incorporating techniques from *genetic programming* and generate shader code that automatically positions samples based on the pixel parameters. Especially depth-of-field would benefit from this extension as the circle of confusion varies per-pixel.

## 6 CONCLUSION

Inspired by the Kawase blur [Kawase 2003], we present a method for finding multi-pass sparse convolution filters that approximate given dense filter masks. Our filters are designed to run

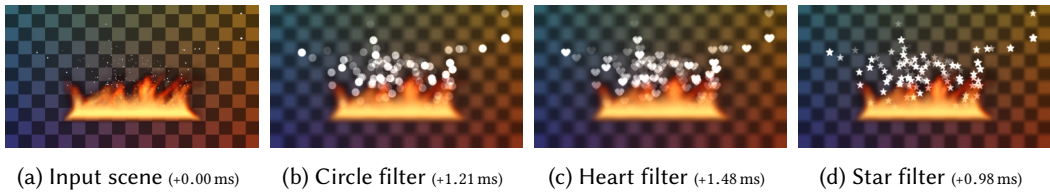


Fig. 12. Screenshots from a dynamic fire simulation scene where the sparks are rendered as bright white particles (a). The full scene is filtered with our approximations of a circle (b), heart (c), and star (d). The scene is rendered in  $1920 \times 1080$  and the render targets have 16 bit resolution. The supplemental material contains a short video of this scene.

on the GPU and exploit the fast bilinear interpolation that the graphics hardware offers. We find these by solving a constrained optimization problem using a modified parallel tempering approach. The optimization target depends on the use case and we present loss functions for a “blur” and a “shape-preserving” case. Our results show that our method consistently finds high-quality approximations that are often more than an order of magnitude faster than the original masks. We even improve upon separated versions, when they exist. We also present a series of novel, high-quality approximations for Gaussian blurs of arbitrary size and various Bokeh shapes. Some of those are shown in Figure 12 and in a short video in the supplemental material. Besides that, the supplemental material contains shader code for all shown filters.

## ACKNOWLEDGMENTS

This work was partially funded by the European Regional Development Fund, within the “Terra Mosana” Interreg Euregio project under the funding code EMR10 and within the “HDV-Mess” project under the funding code EFRE-0500038. Furthermore, this work was partially funded by the German Research Foundation within the Gottfried Wilhelm Leibniz programme under the funding code KO 2064/6-1. Additionally, we are grateful to Sylvia Schuster for the permission to use the photo in Figure 10 and to Julian Schakib for the implementation of the fire simulation in Figure 12.

## REFERENCES

- Marius Bjørge. 2015. Bandwidth-efficient Graphics. In *ACM SIGGRAPH 2015 Courses - Moving Mobile Graphics (SIGGRAPH '15)*. Association for Computing Machinery, New York, NY, USA, Article Article 18, 1 pages.
- John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), 679–698.
- Robert L Cook. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics (TOG)* 5, 1 (1986), 51–72.
- Phill Djonov. 2012. Bicubic Filtering in Fewer Taps. *Shiny Pixels* (2012). <http://vec3.ca/bicubic-filtering-in-fewer-taps/>
- Ondrej Fialka and Martin Cadik. 2006. FFT and convolution performance in image filtering on GPU. In *Tenth International Conference on Information Visualisation (IV'06)*. IEEE, 609–614.
- Kleber Garcia. 2017. Circular separable convolution depth of field. In *ACM SIGGRAPH 2017 Talks*. 1–2.
- Rafael C Gonzales and Richard E Woods. 2018. *Digital Image Processing* (4 ed.). Pearson. 153–162 pages.
- Craig Gotsman. 1994. Constant-Time Filtering by Singular Value Decomposition. In *Computer Graphics Forum*, Vol. 13. Wiley Online Library, 153–163.
- Peter J Green. 1995. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika* 82, 4 (1995), 711–732.
- Christopher G Harris, Mike Stephens, et al. 1988. A combined corner and edge detector.. In *Alvey vision conference*, Vol. 15. Citeseer, 10–5244.
- Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. 2005. Fast summed-area table generation and its applications. In *Computer Graphics Forum*, Vol. 24. Wiley Online Library, 547–555.
- Hyuntae Joo, Soonhyeon Kwon, Sangmin Lee, Elmar Eisemann, and Sungkil Lee. 2016. Efficient ray tracing through aspheric lenses and imperfect bokeh synthesis. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 99–105.

- Masaki Kawase. 2003. Frame Buffer Postprocessing Effects in DOUBLE-STEAL (Wrechless). In *Game Developers Conference 2003*, 3.
- Todd Jerome Kosloff, Justin Hensley, and Brian A Barsky. 2009. Fast filter spreading and its applications. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-54* (2009).
- Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. 2009a. Depth-of-field rendering with multiview synthesis. *ACM Transactions on Graphics (TOG)* 28, 5 (2009), 1–6.
- Sungkil Lee, Elmar Eisemann, and Hans-Peter Seidel. 2010. Real-time lens blur effects and focus control. *ACM Transactions on Graphics (TOG)* 29, 4 (2010), 1–7.
- Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. 2008. Real-time depth-of-field rendering using point splatting on per-pixel layers. In *Computer Graphics Forum*, Vol. 27. Wiley Online Library, 1955–1962.
- Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. 2009b. Real-time depth-of-field rendering using anisotropically filtered mipmap interpolation. *IEEE Transactions on Visualization and Computer Graphics* 15, 3 (2009), 453–464.
- Josiah Manson and Scott Schaefer. 2013. Cardinality-constrained texture filtering. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–8.
- Josiah Manson and Peter-Pike Sloan. 2016. Fast filtering of reflection probes. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 119–127.
- Pavlos Mavridis and Georgios Papaioannou. 2011. High quality elliptical texture filtering on GPU. In *Symposium on Interactive 3D Graphics and Games*. 23–30.
- Joel McCormack, Ronald Perry, Keith I Farkas, and Norman P Jouppi. 1999. Feline: fast elliptical lines for anisotropic texture mapping. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*. 243–250.
- Tim McGraw. 2015. Fast Bokeh effects using low-rank linear filters. *The Visual Computer* 31, 5 (2015), 601–611.
- L McIntosh, Bernhard E Riecke, and Steve DiPaola. 2012. Efficiently Simulating the Bokeh of Polygonal Apertures in a Post-Process Depth of Field Shader. In *Computer Graphics Forum*, Vol. 31. Wiley Online Library, 1810–1822.
- Johannes Moersch and Howard J Hamilton. 2014. Variable-sized, circular bokeh depth of field effects. In *Proceedings of Graphics Interface 2014*. 103–107.
- Diego Nehab and André Maximo. 2016. Parallel recursive filtering of infinite input extensions. *ACM Transactions on Graphics (TOG)* 35, 6 (2016), 1–13.
- Diego Nehab, André Maximo, Rodolfo S. Lima, and Hugues Hoppe. 2011. GPU-Efficient Recursive Filtering and Summed-Area Tables. *ACM Trans. Graph.* 30, 6 (Dec. 2011), 1–12.
- Dan Piponi. 2012. Fast and Exact Convolution with Polygonal Filters. (01 2012).
- Michael Potmesil and Indranil Chakravarty. 1982. Synthetic image generation with a lens and aperture camera model. *ACM Transactions on Graphics (TOG)* 1, 2 (1982), 85–108.
- Daniel Rákos. 2010. Efficient Gaussian blur with linear sampling. *RasterGrid Blogosphere* (2010). <http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/>
- Christian Sigg and Markus Hadwiger. 2005. Fast third-order texture filtering. *GPU gems 2* (2005), 313–329.
- Tiago Sousa. 2013. CryEngine 3 Graphics Gems. *ACM SIGGRAPH Advances in Real-Time Rendering Course* (2013).
- Filip Strugar. 2014. An investigation of fast real-time GPU-based image blur algorithms. (2014). <https://software.intel.com/en-us/blogs/2014/07/15/an-investigation-of-fast-real-time-gpu-based-image-blur-algorithms>
- Swendsen and Wang. 1986. Replica Monte Carlo simulation of spin glasses. *Physical review letters* 57 21 (1986), 2607–2609.
- Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.
- John White and Colin Barré-Brisebois. 2011. More Performance! Five Rendering Ideas from Battlefield 3 and Need For Speed: The Run. *ACM SIGGRAPH 2011: Advances in the realtime rendering course* (2011).