DOI: 10.1111/cgf.14049
Eurographics Symposium on Rendering 2020
C. Dachsbacher and M. Pharr
(Guest Editors)

Volume 39 (2020), Number 4

Neural Denoising with Layer Embeddings

Jacob Munkberg & Jon Hasselgren

NVIDIA

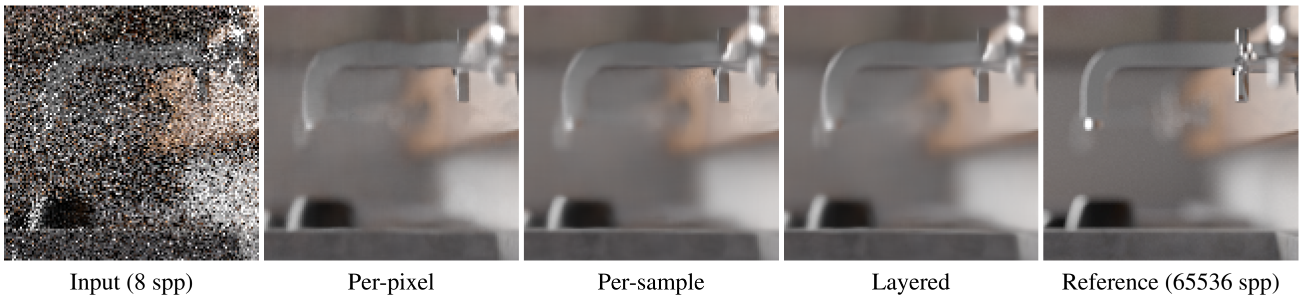

Figure 1: *Neural denoisers for Monte Carlo path tracing provide impressive reconstruction from low sample counts. However, in challenging scenarios, with significant noise in* both *shading and visibility, denoisers working on accumulated pixel values sometimes struggle. Our layered denoiser improves image quality over per-pixel denoisers at a fraction of the computational cost of a per-sample denoiser.*

Abstract
We propose a novel approach for denoising Monte Carlo path traced images, which uses data from individual samples rather than relying on pixel aggregates. Samples are partitioned into layers, which are filtered separately, giving the network more freedom to handle outliers and complex visibility. Finally the layers are composited front-to-back using alpha blending. The system is trained end-to-end, with learned layer partitioning, filter kernels, and compositing. We obtain similar image quality as recent state-of-the-art sample based denoisers at a fraction of the computational cost and memory requirements.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Ray tracing; Neural networks;***

1. Introduction

Monte Carlo (MC) path tracing is a rendering technique to accurately simulate light transport in computer graphics, by tracing multiple ray paths, here denoted samples, within each pixel. For a noise-free radiance estimate, 10k–100k samples per pixel (spp) are not uncommon, depending on the complexity of the scene and lighting configuration. To remove residual noise from this stochastic rendering process, a common technique is to filter, or denoise, the image, by gathering samples in a spatial neighborhood around each pixel, often with edge-preserving filter kernels. This process introduces some bias but can reduce residual noise to acceptable levels, even from low sample counts, e.g., 8–64 spp. Denoising is most often applied to the accumulated pixel values, but recent work shows that by denoising from individual samples, reconstruction quality can be improved, but at an increased computational cost [GLA*19].

In this article, we evaluate the benefits of using per-sample information over pixel aggregates in a machine learning denoiser for Monte Carlo (MC) path tracing. We are seeking compact representations to capture the information discovered by individual path samples. There are two extremes: either keeping all individual samples or compressing them to a single value, e.g., the mean, or moments (e.g. mean and variance). We carefully analyze and evaluate the quality differences of these approaches within a single framework. In our evaluation, the per-sample denoiser is more robust against high intensity outliers, but the quality differences are subtle.

Per-sample denoisers come with high computational costs; denoising times are measured in seconds or minutes. Both runtime performance and memory scale linearly with sample count. The main computational cost stems from the fact that the network need to produce kernel weights and apply a large kernel for *each sample in each pixel*. Paradoxically, this means that resource consumption grows as input quality increases, even though the denoising task becomes easier with higher-quality input. This somewhat limits usefulness as per-sample denoisers are too expensive for real-time rendering, and may not scale up to the sample counts required for photo-realistic production rendering. In this work, we strive to extract a compact representation of per-sample information. We take data-driven approach, and let a neural network *learn* this representation. We constrain the denoiser to use a fixed number of partitions, that we denote layers. This results in a middle ground: a practical denoiser that strikes a good trade-off between cost and quality.

We propose a novel architecture that learns to partition samples into layers, learns unique kernels weights for each pixel in each layer, and learns how to composite the filtered layers. This gives us similar benefits of a recent per-sample denoising architecture of Gharbi et al. [GLA*19], but with considerably better performance characteristics. The partitioning of samples into layers provides additional flexibility for the network to robustly handle outliers/fireflies and cases with complex primary visibility, including defocus blur and motion blur. Furthermore, it gives us an efficient way to control performance and memory characteristics, as our algorithm scales with the number of layers rather than the number of samples. In our evaluation, we note that two layers are sufficient to get most of the benefits of a per-sample denoiser. Additionally, we design an efficient, pruned, denoiser that runs at interactive rates while producing image quality similar to the larger networks.

The main contributions of this work are:

- Robust denoising by exploiting sample information from the renderer instead of only pixel aggregates.
- A novel layered denoising architecture with learned layer partitions, filters and alpha compositing.
- An analysis of different layer partitioning strategies.
- Significant runtime performance improvements over previous sample-based denoisers.

2. Previous Work

There has been substantial recent progress in denoising for offline rendering. A great overview is available in the survey from Zwicker et al. [ZJL*15]. Below, we will mainly discuss machine learning (ML) denoising techniques. Recent ML denoisers use large convolutional neural networks (CNN) to reconstruct noisy Monte Carlo renderings with the help from additional feature guides (surface normals, depth, albedo, . . .). Xu et al. [XZW*19] use feature guides combined with an adversarial loss function to retain more high-frequency details. Wong and Wong [WW19] let a deep residual network directly predict the denoised colors.

Several recent denoisers let the network predict unique filter kernels per pixel, for single frames [BVM*17] or time sequences [VRM*18], by running the network in parallel over multiple frames. The learned kernels are applied to filter the noisy input as a *gather* operation. In contrast, Gharbi et al. [GLA*19] apply kernels as a *splatting* operation of pixel aggregates or individual samples onto the framebuffer. Leimkuhler et al. [LSR18] use Laplacian kernel splatting to reconstruct defocus and motion blur.

Kernel prediction (KP) improves robustness, as the same filter weights are applied to the each color channel (reduced color shifts), and it constrains the filtered output to be a linear combination of the input samples. It has also been successfully applied to denoise bursts of camera images [MBC*18] and video interpolation [NML17b, NML17a].

We are primarily inspired by the recent work of Gharbi et al. [GLA*19] who apply kernel based denoising using CNNs on *individual samples*, typically working with 8–32 samples per pixel. They learn sample embeddings, process the (pixel-averaged) embeddings by U-nets [RFB15], combine them with per sample data to generate filter weights, and finally splat large per-sample kernels onto the framebuffer. They show high quality results on still images, albeit at a high computational cost with denoising times measured in seconds or minutes. Similarly, memory requirements are substantial with 74 scalar features per rendered sample. Both performance and memory scales linearly with sample count. The algorithm is too expensive for real-time rendering, while it may not scale up to the sample counts required for production rendering. In this work, we adapt a layered architecture to design an efficient denoiser with near real-time performance, while retaining most benefits of per-sample denoisers.

The machine learning reconstruction approaches for Monte Carlo rendering discussed above generate state-of-the-art denoising quality from 8–64 samples per pixel (spp), but come at significant cost: seconds per frame on a high end GPU. For real-time reconstruction, smaller U-Nets that directly predict the output colors instead of KP, have been applied [CKS*17, LMH*18]. To improve runtime performance of KP, Vogels et al. [VRM*18] propose to approximate a single large kernel with a hierarchy of smaller kernels, each applied on spatially down-sampled versions of the input image. They mention that recent experiments with a multi-scale hierarchy of 5×5 kernels, using a kernel prediction at the end of each U-Net decoder scale, generates good results at improved performance. Hasselgren et al. [HMP*20] use a similar hierarchy of filters to reach interactive denoising performance. We adapt a three-level hierarchy in this paper to accelerate our real-time denoiser.

There is a vast body of work on using layers to help reconstruct defocus and motion blur from individual samples. Zimmer et al. [ZRJ*15] propose a general decomposition framework, where the final pixel color is separated into components corresponding to disjoint subsets of the space of light paths. Layered light field reconstruction for defocus and motion blur [VMCS15, MVH*14] partition samples front-to-back into depth layers. The depth layers are individually filtered, then composited using alpha blending. Bauszat et al. [BEJM15] extend adaptive manifold filtering [GO12] to work on individual samples and apply the technique to simultaneous defocus blur and Monte Carlo-simulated global illumination at interactive rates. They use a set of depth layers, splat samples to layers using the sample's circle of confusion radius, filter each layer separately, and composite the result. We have a similar goal of fast denoising from low sample counts by exploiting per-sample information and layers, but we apply neural networks to learn both the layer partitioning and denoising kernels. In addition, we denoise the more general case of simultaneous motion and defocus blur combined with the noise from Monte Carlo path tracing.

Our work is also related to deep framebuffer techniques, wherein pixels contain multiple color values, each for a different range of depths. Vicini et al. [VAN*19] present techniques for efficient denoising approaches on deep images. We see potential for adapting our approach to that domain, given that deep compositing workflows produce sample partitions similar to our layers.

3. Architecture

Our denoiser architecture is shown in Figure 2. The first half is similar to the per-sample architecture by Gharbi et al. [GLA*19], and the second half introduces our layer embedding stages. We target a denoiser running at interactive rates, and want to assess the value of a per-sample network in that setting. To that end, compared to the network of Gharbi et al., we reduce the number of input features from 74 to 20 and use a single larger U-net instead of a sequence of three U-nets. Carefully pruning the feature count and the network size is a worthwhile research topic in itself, but outside the scope of this paper.

Conceptually, our architecture can be split up into five main stages:

Sample reducer Input sample radiance and feature guides are reduced through a network to generate *sample embeddings*.

U-net Sample embeddings are averaged and processed in a U-net [RFB15] with large spatial footprint. This network outputs *context* features.

Sample partitioner Sample embeddings and radiance are splat into two or more layers, creating *layer embeddings*. Splatting weights are computed from sample embeddings and context features, using a small fully connected network.

Layer filtering A per layer spatial filter is applied, with unique filter kernels for each pixel. Filter kernel weights are computed by a fully connected network, taking layer embeddings and context features as input.

Compositing We track both radiance and weight/opacity per layer. The final output is obtained by compositing the filtered layers.

Pseudocode for the architecture is included in Appendix A.

Per-sample Processing. The input to our architecture is per sample radiance and feature guides, computed by a Monte Carlo path tracer. We also provide *normal*, *depth*, *albedo*, *specular color*, *roughness*, *motion vector*, *circle of confusion*, *lens position*, and *time*, for a total of 20 floats per sample. It should be noted that this is a smaller set than what was used in the work by Gharbi et al. [GLA*19], as we target interactive denoising.

The first stage is the **Sample reducer** stage of Gharbi et al. It takes radiance estimates, L_{xys}, of sample s at pixel coordinates x and y and corresponding per-sample feature guides f_{xys} and produces sample embeddings, E_{xys}. This is accomplished by applying a small fully connected network on each sample. Our sample embeddings have 32 features.

Per-pixel Processing. We compute *context* features by averaging sample embeddings per pixel and process them by a large **U-net** [RFB15] with high feature counts and large spatial footprint. This design allows the network to gather data and perform feature detection on a spatially large set of input samples. The output of the U-net is 128 context features, C_{xy}, which are used further down the architecture pipeline, both to partition samples into layers and to generate filter kernels. The exact layout of our U-net is shown in Figure 3.

In the second part of our architecture, we partition the samples into layers and then filter and process each layer individually. Both layer partitioning and filtering are handled by networks, and the entire system is trained end-to-end. Below, we describe the second part in more detail.

Per-layer Processing. To partition samples into layers, we splat a (learned) fraction of each sample to each layer. As shown in Figure 2, we use a fully connected network, the **Sample partitioner**, which operates on sample embeddings, E_{xys}, and context features, C_{xy}, and outputs a scalar weight per layer, l, and sample, s:

$$w^l_{xys}, \qquad \sum_l w^l_{xys} = 1. \tag{1}$$

We then splat the sample embedding and radiance of each sample into each layer, giving us *layer embeddings* for each layer l.

To filter and composite the layers, we use alpha-blending, as it yielded better quality than a straightforward weighted sum of filtered layers. Compositing alternatives are discussed in more details in Section 3.1. More precisely, we extend a layered light field reconstruction technique [VMCS15] to use learned filter kernels and layer partitioning. Samples are grouped front-to-back into depth layers. The depth layers are individually filtered and composited using alpha blending. We track, for each layer, the weight sum w^l_{xy}, layer occupancy n^l_{xy}, and weighted sums of radiance L^l_{xy} and embeddings E^l_{xy}:

$$\begin{aligned} w^l_{xy} &= \sum_s w^l_{xys} \\ n^l_{xy} &= \sum_s \sum_{k=0}^{l-1} w^k_{xys} \\ L^l_{xy} &= \sum_s w^l_{xys} L_{xys} \\ E^l_{xy} &= \sum_s w^l_{xys} E_{xys}. \end{aligned} \tag{2}$$

When a sample is written to a layer, it *punches a hole* in all layers in front of it, indicating that these layers are transparent at that particular position. The layer occupancy buffer, n^l_{xy}, tracks this information. This approach enables accurate layer opacity estimates from low sample counts. For more details about this compositing approach, please refer to Vaidyanathan et al. [VMCS15].

We then proceed to filter each layer. We follow recent kernel predicting denoising networks [VRM*18], and compute, for each layer, kernel weights, K^l_{xyuv}, for a spatial filter, unique for each pixel. The kernel weights are computed using a fully connected **Kernel generator** network, operating on layer embeddings, E^l_{xy}, and context features C_{xy}. Following Gharbi et al. [GLA*19], we apply the kernels as *splatting* operations, as this helps with suppressing fireflies. We apply (denoted with $*$) the kernel to the layer radiance, L^l_{xy}, weights, w^l_{xy}, and occupancy, n^l_{xy}, to produce filtered

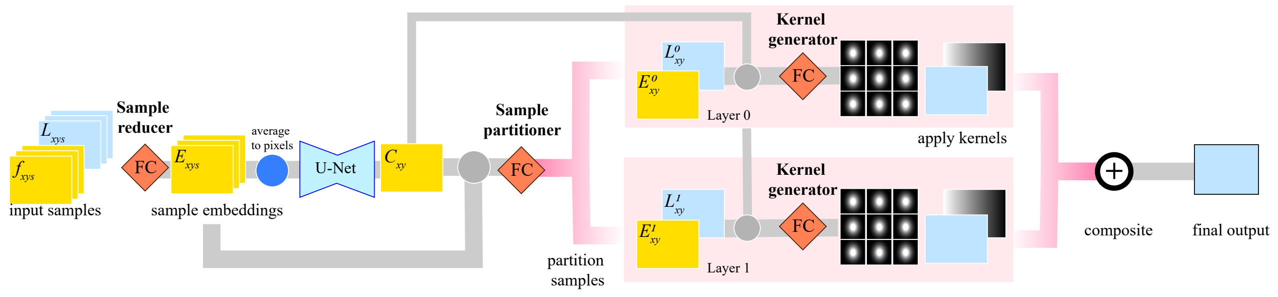

Figure 2: *Network architecture overview, adapted from Fig. 3 in Gharbi et al. [GLA*19]. The network receives individual radiance samples L_{xys} with corresponding features f_{xys} as input. The* ***Sample reducer*** *network generates sample embeddings E, average them per pixel, and feed the averages into a* ***U-Net*** *(see Figure 3 for an expanded view) to provide context features C at pixel rate. Then, a fully connected* ***Sample partitioner*** *network takes the context features C_{xy} and sample embedding E_{xys} to produce scalar weights, $w^l_{xys} \in [0,1]$, which represent the fraction of the sample to assign to each layer. We then partition the input radiance samples L and sample embeddings E into clusters per layer L^l_{xy} and E^l_{xy}. Each cluster represents the weighted average of the samples assigned to that layer. Given the per-layer clusters, we run a fully connected* ***Kernel generator*** *network to produce a unique filter kernel per pixel in each layer from the layer embedding and the context features C_{xy}. This kernel is applied to the linear radiance samples L^l_{xy} and layer weights w^l_{xy} to produce a filtered layer radiance estimate (blue square), and a filtered layer opacity (gradient square). Finally, the layers are composited to form the output image.*

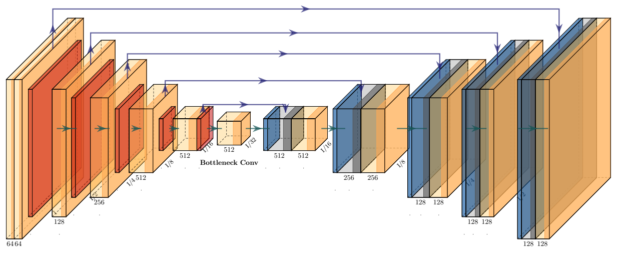

Figure 3: *Our U-net. We use convolution kernel sizes of 3×3, leaky ReLU activation functions with a negative slope of 0.01, bilinear upsampling and 2×2 max-pooling. The skip connections are concatenated with the input features.*

versions:

$$\begin{aligned} \bar{L}^l_{xy} &= L^l_{xy} * K^l_{xyuv} \\ \bar{w}^l_{xy} &= w^l_{xy} * K^l_{xyuv}. \\ \bar{n}^l_{xy} &= n^l_{xy} * K^l_{xyuv}. \end{aligned} \tag{3}$$

Compositing. We compute the final output in this step. We first normalize radiance and define α as the normalized layer weight

$$\hat{L}^l_{xy} = \frac{\bar{L}^l_{xy}}{\bar{n}^l_{xy}}, \quad \alpha^l_{xy} = \frac{\bar{w}^l_{xy}}{\bar{n}^l_{xy}}. \tag{4}$$

The compositing operation to obtain the final pixel color o_{xy} is then:

$$o_{xy} = \hat{L}^0_{xy} + \sum_{l=1}^{N} \hat{L}^l_{xy} \prod_{j=0}^{l-1} (1 - \alpha^j_{xy}), \tag{5}$$

e.g., a front-to-back alpha compositing with pre-multiplied values.

Please refer to Appendix A for pseudocode for the forward pass of our architecture, and further network implementation details in Appendix B. Although our architecture is divided into several stages, we want to stress that the system is trained end-to-end, optimizing only for the error of the final output, so all steps are optimized jointly.

3.1. Discussion

Scaling costs. Comparing our approach to Gharbi et al., the sample reducer and sample partitioning steps scale linearly with the number of samples, while the kernel generation and composite steps scale with the number of layers. A key observation is that the sample reducer and sample partitioner networks are comparatively inexpensive compared to the kernel generation step. The kernel generator in a per-sample network outputs all weights for a large filter kernel per sample. A 21×21 kernel per sample results in $21 \times 21 \times 8 = 3528$ features for each pixel at 8 spp. With two layers, we reduce this cost by a factor of four.

By moving to a layered representation, we can eliminate the main bottleneck of the system, replacing the per-sample multiplier with a *layer count* parameter, which can be used to balance quality and performance. We evaluate the computational cost of the different network in Section 6.

Compositing Alternatives. One argument for using per-sample data is that effects such as defocus blur and depth of field cannot be robustly handled with per-pixel data [GLA*19]. In scenarios such as a moving sphere in front of a static background or out-of-focus foreground objects over an in-focus background (Figure 4), the samples can naturally be partitioned into depth layers with unique filters per layer. This motivated our trainable alpha compositing as described in the previous section.

An alternative, more general, compositing approach is to treat each layer as a weighted sum of samples, with no notion of ordering

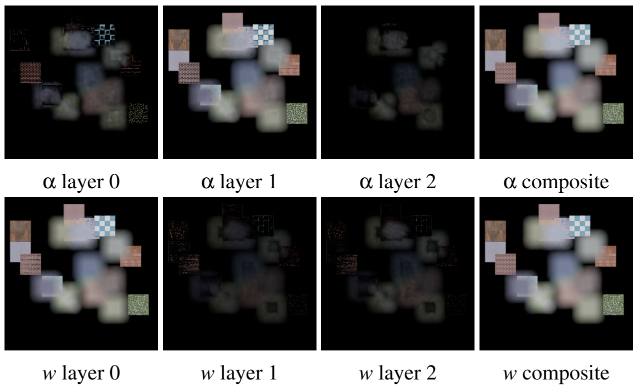

Figure 4: *A synthetic test with an out-of-focus foreground layer and an in-focus background layer. We train two networks, one with ordered alpha blending compositing, denoted α, the other with weighted layers (w). The partitioning of samples into layers and layer kernels are learned. We visualize the filtered layers and the composited result. Comparing the two approaches, we see that the α-version learns roughly a depth-based partitioning, with the in-focus background layer in the middle layer, but splits texture details across the layers. The weight based approach puts most information in one layer, and uses the other layers to separate frequencies, somewhat similar to a wavelet decomposition.*

or visibility between layers. In this case, there is no need for the occupancy buffer, and the compositing operation is simply

$$o_{xy} = \frac{1}{\sum \bar{w}^l_{xy}} \sum_{l=0}^{N} \bar{w}^l_{xy} \bar{L}^l_{xy}. \quad (6)$$

Please compare this to the alpha compositing in Equation 5.

In Figure 4 we compare the two compositing approaches in a synthetic experiment, where the networks are trained to denoise rectangular textured primitives at different depths under defocus blur. We eliminated all noise but visibility noise from the lens sampling. The opacity-based layers with strict ordering and alpha blending are intentionally designed to encourage the network to partition samples based on depth. The results do not show a strict depth partitioning, but we note that the alpha network keeps mostly foreground defocus regions in layer 0, and the in-focus background objects are concentrated to layer 1.

We then trained networks variants with the two composting techniques on a large corpus of path traced images. In our experience, the alpha compositing approach produces slightly higher-quality results than the weighed approach, both visually (please refer to the supplemental material) and in the image quality metrics (see Table 1). We believe the more constrained approach may help the network converge to a better local minimum, similar to how kernel prediction seems to work better than direct prediction, although the latter approach is more general. The runtime costs are similar. The alpha-compositing approach require an additional buffer per layer (occupancy, n^l_{xy}).

In Figure 5, we visualize our network (alpha compositing variant) with two layers when trained on our entire dataset. Here we note that the network assigns outlier samples (fireflies) to layer L^0, with layer L^1 looking like a lower variance version of the input image. The network is, broadly speaking, dividing the image into a hard and easy part, applying different filters for each partition. This helps corroborate why layer separation is useful. The take-away is that giving the network freedom to partition samples into two or more layers is beneficial, the choice of compositing operation less so. For the remainder of this paper, results are presented for the α-version unless otherwise noted.

4. Dataset and Training

We generate a large collection of scenes to train our networks. We follow the random scene generation approach presented by Gharbi et al. [GLA*19] and use 50,000 3D models from ShapeNet [CFG*15], 40 HDR probes from HDRI Haven [Zaa16], 5800 textures from the Descriptable Texure Dataset [CMK*14] and the Pixar 128 textures [Sis16].

The outdoor scenes have 75 random meshes on a backdrop mesh, lit with an HDR probe. For our indoor scenes, we use one of eight simple rooms filled with 50 random meshes, lit with four area lights, one point light and a random HDR probe. All lights are given random positions and radiance. Likewise, camera parameters such as position, field-of-view and defocus blur settings are randomized. We assign each mesh a randomized material from a large set of BSDFs (diffuse, microfacet, dielectric, mirror, conductor, coated, etc.), randomize material parameters and textures and randomly scale texture coordinates to vary texture frequency. For frames with motion blur, we additionally assign a random, linear, object motion with 50% probability. Each scene is independently selected to include motion or defocus blur with 50% probability. Some example scenes are shown in Figure 6.

The scenes are MC path traced with five bounces and Russian Roulette disabled. We render input frames at 8 spp and store per-sample radiance and guides (20 floats per sample). References are rendered at 4096 spp and only average radiance per pixel is stored. We initialize the renderer with different random number seeds in the input and references to avoid correlations. Each training example is rendered at a resolution of 256×256 pixels. In total the training set consists of 4352 frames, of which half are indoor and half are outdoor scenes.

We implemented our networks in PyTorch [PGC*17]. We use Xavier initialization [GB10] and train with Adam [KB15] with default parameter settings and an initial learning rate of 0.005. We train with input augmentations (random crops, flip x/y, 90 degree rotations) and shuffle the training data each epoch. We use spatial crops of size 128×128 and batch size four. All networks are trained for 1000 epochs unless otherwise noted, so in a typical training session, the networks see 4.3 million different image pairs. We also track a small validation set of 128 additional randomly generated frames, to supervise the training progress. Training a network for 1000 epochs with the augmented training dataset takes between 30 and 60 hours on a single NVIDIA V100 GPU. The per-sample network is, not surprisingly, the slowest to train. The kernels produced by the network is applied on linear radiance values, and the final network output are linear radiance values.

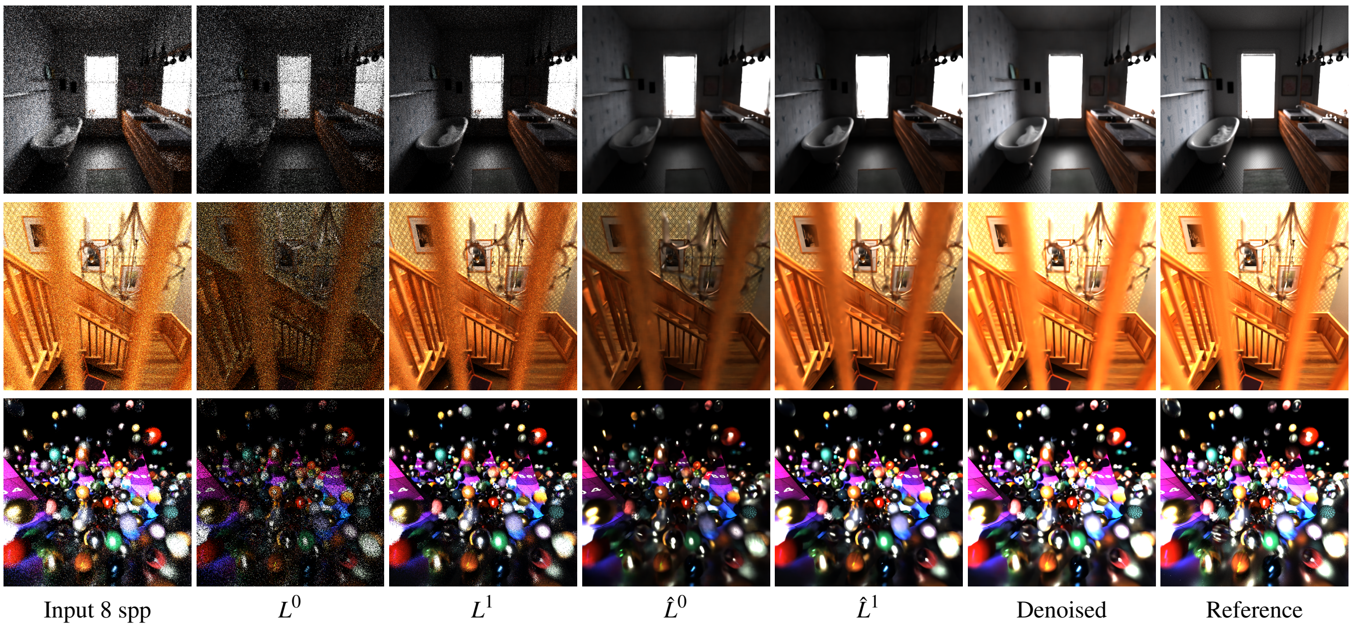

Figure 5: *From left to right: input samples, unfiltered layer radiance L^i (Equation 2), filtered layer radiance, $\hat{L}^i$ (Equation 4), the composited result (Equation 5), and the reference. We visualize all images in linear space without tone mapper to highlight absolute intensity differences. Note that the network learns to partition high variance samples into one layer.*

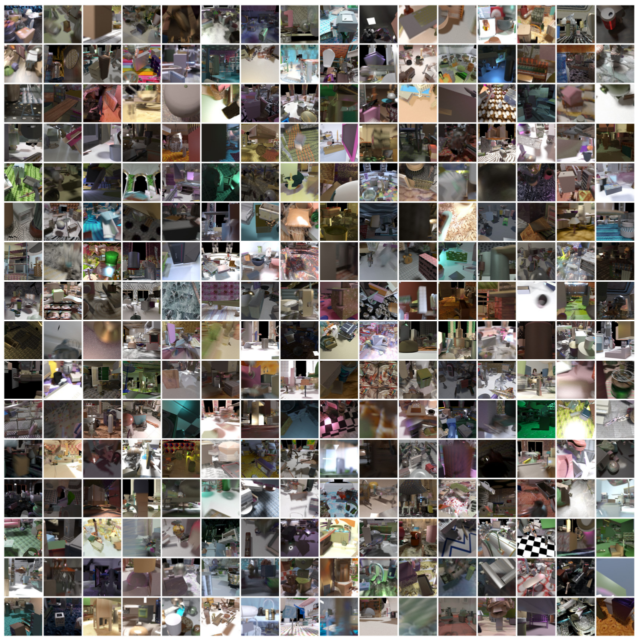

Figure 6: *Example training data from the random scene generator.*

We trained the network with two different loss functions. First, following Gharbi et al., we use L_2 loss on tone mapped values as optimization criterion. As tone map operator, we transform linear radiance values, x, according to

$$x' = \Gamma(\log(x+1)), \tag{7}$$

where $\Gamma(x)$ is the sRGB transfer function [SACM96]:

$$\begin{aligned} \Gamma(x) &= \begin{cases} 12.92x & x \leq 0.0031308 \\ (1+a)x^{1/2.4} - a & x > 0.0031308 \end{cases} \\ a &= 0.055. \end{aligned} \tag{8}$$

We found this tone mapper to be more robust in training than the Reinhard tone mapper used in Gharbi et al. A similar observations was made in a recent machine learning algorithm for burst denoising [MBC*18]. Second, we tried the *symmetric mean absolute percentage error* (SMAPE), which has been reported to be stable when denoising HDR images [VRM*18]. Given a reference $\mathbf{r}$ and denoised estime $\mathbf{d}$:

$$SMAPE(\mathbf{r},\mathbf{d}) = \frac{1}{3N} \sum_{p \in N} \sum_{c \in C} \frac{|\mathbf{d}_{p,c} - \mathbf{r}_{p,c}|}{|\mathbf{d}_{p,c}| + |\mathbf{r}_{p,c}| + \varepsilon}. \tag{9}$$

Here, N is the number of pixels in the image, and C are the three color channels. We use $\varepsilon = 0.01$. When comparing the network trained with the two loss functions, we observed better results with SMAPE, both by visual inspection and in the error metrics. We include a comparison in Appendix C.

5. Image Quality Evaluation

In this section, we focus on the image quality differences of per-sample, per-pixel and layered denoisers in similar settings. It is now well understood how to craft a high quality machine

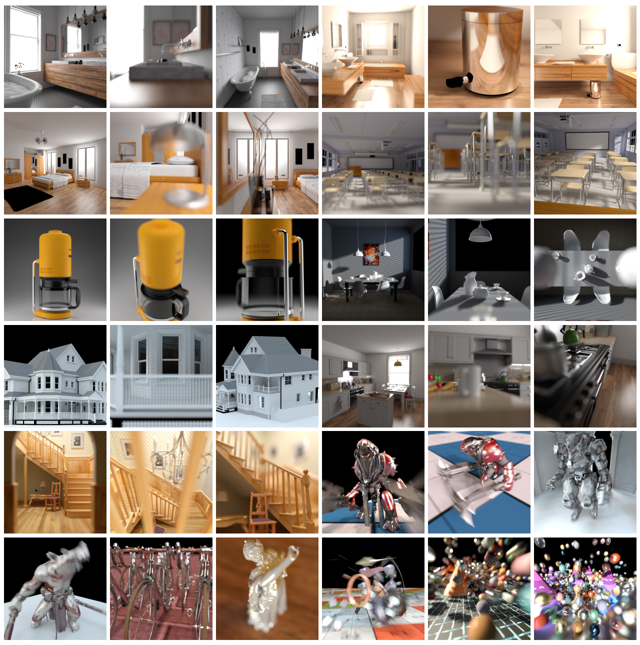

Figure 7: *Our test set. We use different viewpoints per scene to capture a larger range of shading and geometry effects. Each frame has different camera settings, aperture and focal length. We also include a few scenes with motion blur with skinned characters and simple physics simulations.*

learning denoiser. See for example the excellent paper by Vogels et al. [VRM*18] that discusses many challenges in designing a denoiser for offline production rendering. For extensive comparisons with other recent denoising techniques, please refer to previous work [BVM*17, VRM*18, GLA*19]. In our experience, the quality of machine learning denoisers are greatly influenced by the training dataset, so in all comparisons in this paper we train all network variants at equal number of steps on the same dataset and loss (SMAPE : Equation 9), to focus on architectural differences and network capacity. We have spent considerable effort building a reasonable large and varied training dataset for the comparisons below. Please refer to Section 4 for details.

To study differences between per-sample, per-pixel and layer networks, we compare three networks. Referring to Figure 2, the first half of the architecture is shared by all configurations. Input samples are transformed into sample embeddings and a U-net (Figure 3) transforms the average embeddings into context features. In the second half, when applying the predicted filter kernels, the variants diverge. We use 17×17 pixel filter kernels for all networks, but they are applied at different granularity:

PIXELGATHER The network outputs one kernel per pixel, which is then applied on the pixel aggregate radiance. The kernel is applied as a gather operation. This variant is similar to the PixelGather network from Gharbi et al. [GLA*19] and Bako et al. [BVM*17] (without albedo demodulation).

SAMPLESPLAT The network outputs a kernel per-sample. The kernel is applied per sample as a splatting operation. This variant is similar to Gharbi et al. [GLA*19].

LAYER Our proposed variant, as depicted in Figure 2 with two layers and unique filter kernels per pixel in each layer.

	relMSE ↓	**SMAPE ↓**	**SSIM ↑**	**PSNR ↑**
PIXELGATHER	0.0352	0.0353	0.938	33.706
SAMPLESPLAT	0.0298	0.0344	0.941	33.878
LAYER$_\alpha$	0.0288	0.0350	0.941	33.838
LAYER$_w$	0.0295	0.0350	0.940	33.812

Table 1: *Image quality metrics. Average over the 36 images in the test set. The relMSE and SMAPE scores are computed on linear radiance values. Lower values means lower error. SSIM and PSNR are computed on tone mapped values, using the tone map operator of Equation 7. Higher values means better quality.*

Note that we have purposefully chosen this setup to keep input and the majority of the networks identical. Thus we isolate and evaluate the impact of *layer embeddings* compared to per-sample or pixel aggregates, and do not compare to the exact implementations of previous work. We argue that this is a fair comparison when evaluating the impact of per-sample information and layers over networks working on pixel aggregates. The size of the U-net, kernel sizes and number of features in our networks differ from previous work. Our network sizes are purposely chosen fairly large to focus on quality and to simplify the comparison of per-sample, layer and per-pixel networks in similar settings.

We evaluate the fully trained networks by denoising the 36 images shown in Figure 7 from 8 spp input. Our test set consists of scenes from Benedikt Bitterli [Bit16], ORCA [NVI18], Smithsonian [Smi18] and Morgan McGuire's Computer Graphics archive [McG17]. We added defocus blur and new camera viewpoints to the Bitterli scenes, and include a few scenes with motion blurred skinned characters and some simple physics simulations with large motion variations.

We show quantitative results in Table 1, as well as a few selected image crops in Figure 8. Please refer to the supplemental material for the test set images evaluated on all network variants. We first note that LAYER obtain similar visual quality as SAMPLESPLAT with only two layers. Looking at just image quality metrics, the differences are minor, which is consistent with the results presented by Gharbi et al. [GLA*19]. In our experience, the main benefit of SAMPLESPLAT is robustness against fireflies. This can be seen in the Bathroom and Spheres scene, where we note that PIXELGATHER produces blocky image artifacts.

A single 17×17 kernel is still not large enough to blur an intense firefly, which creates box-shaped artifacts in the denoised image (e.g., second row in Figure 8). LAYER and SAMPLESPLAT have more freedom in suppressing fireflies, and produce smoother, better looking, images. The remaining examples show cases with significant noise both in primary visibility (due to defocus blur), and shading due to complex lighting sampled at only 8 spp. We note that PIXELGATHER again struggles in regions with complex visibility, such as for the Classroom scene, where it loses detail or has remaining low frequency noise, giving it a blotchy look.

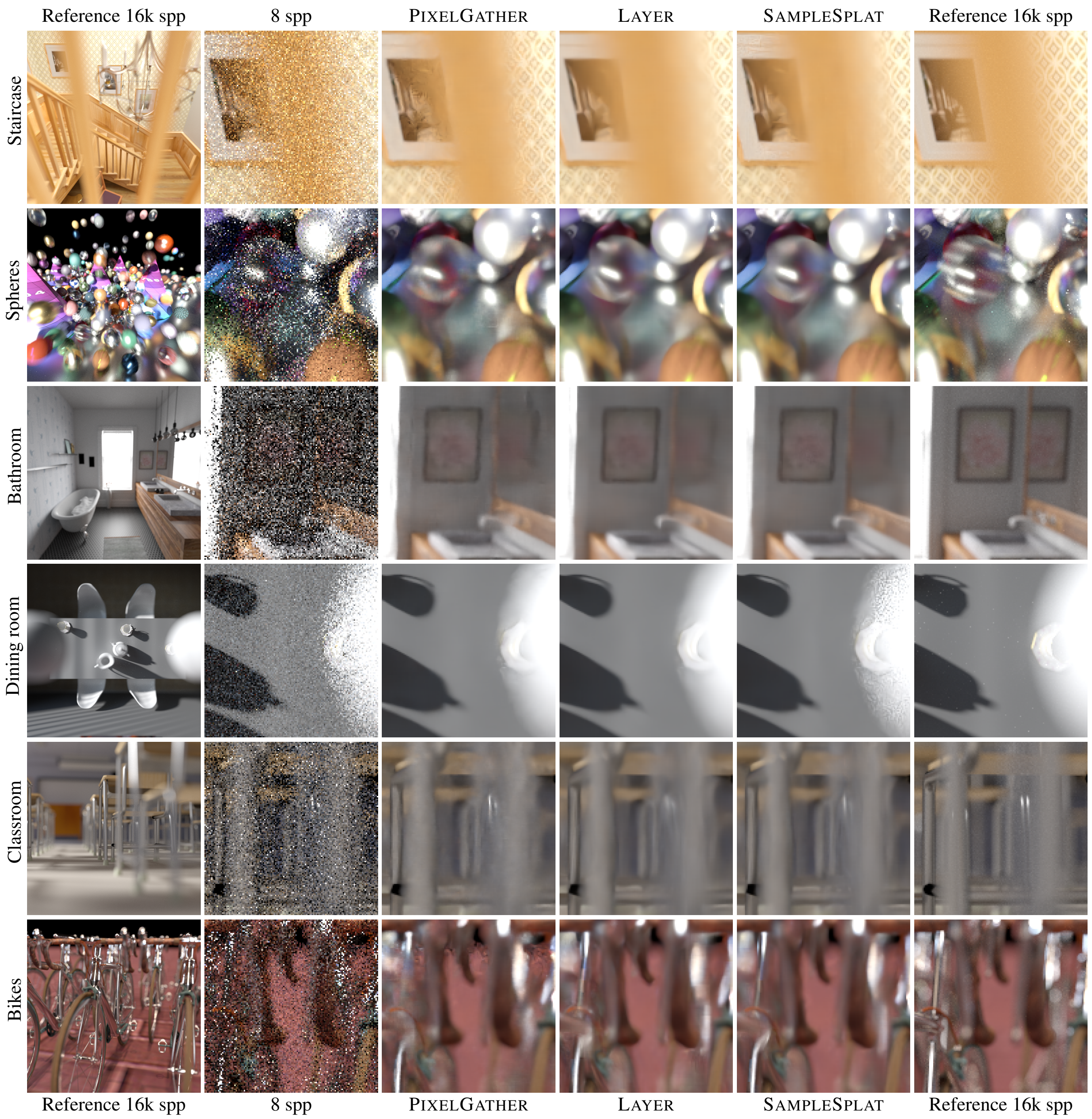

Figure 8: *Denoising quality from 8 spp. A few selected crops from our test set. Please zoom in to see the details, as the differences are subtle. We provide results from all network variants on the test set in the supplemental material for detailed comparisons.*

We study the impact of layer count in Table 2. Increasing layer count above two has limited quality impact and sometimes also reduce quality. When the layer-to-sample ratio is high, the sample partitioning step becomes more complex, and may cause the network to get stuck in a worse local minimum.

6. Optimizations and Runtime Performance Evaluation

In this section, we evaluate optimization strategies to significantly reduce runtime with limited impact on image quality, to approach interactive denoising rates.

To reduce the number of kernel weights produced by the

layers	**relMSE** ↓	**SMAPE** ↓	**SSIM** ↑	**PSNR** ↑
2	0.0288	0.0350	0.941	33.838
3	0.0315	0.0364	0.937	33.611
4	0.0313	0.0361	0.938	33.697
5	0.0332	0.0353	0.939	33.849

Table 2: *Error metrics as function of the number of layers in the network, evaluated on out test set (Figure 7). We note that there is limited benefit of moving beyond two layers.*

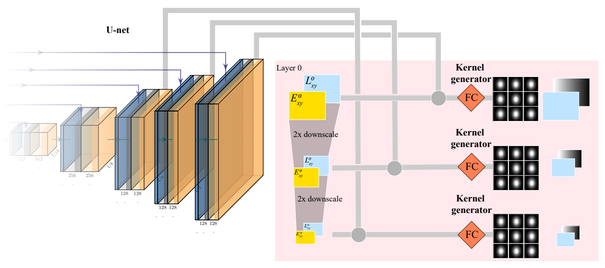

Figure 9: *Example of hierarchical kernel prediction with three levels applied in our architecture. For each layer (we show layer 0), kernel prediction is applied on three scales. The kernel generator network is fed with the scaled layer embedding and the* ***U-net*** *decoder layer at the same resolution.*

network, we apply hierarchical kernel prediction [VRM*18, HMP*20]. Instead of applying a large, 17×17 kernel per pixel per layer, we apply a cascade of three 5×5 filters, applied to the original resolution as well as $2\times$ and $4\times$ downscaled version of the input radiance values. Referring to Figure 9, we feed the kernel generator networks with downscaled versions of the layer embeddings as well as the U-net decoder layer output with corresponding resolution. The generated kernel is then applied on the downscaled radiance.

Kernel normalization is more complicated when using hierarchical kernels, as weights are applied at different spatial resolutions. As a practical solution, we process each miplevel until completion. That is, we perform layer filtering for all layers, and do compositing to get a *global* miplevel, G_i, which can be seen as the final, correctly normalized, output of our algorithm as computed at given sub-sampled resolution. To get the final result, we combine the global miplevels using another weighted summation,

$$\sum_{i=0}^{2} \text{Upscale}(G_i, 2^i)\lambda_i, \qquad (10)$$

where λ_i are trainable scalar weights. We modify the kernel generator network of the first layer to additionally output the three λ_i weights.

Hierarchical kernels result in optimized filter evaluations, but, more significantly, reduced amount of data flowing through the network. A 17×17 filter requires 289 features, which is the reason our U-net and kernel generator network are sized accordingly with high feature counts. The hierarchy allows us to reduce the kernel generator network from 128 to 32 features throughout, thus significantly reducing one of the main architectural bottlenecks. We similarly reduced the U-net (Figure 3) feature count in the first two convolutional layers of the encoder to 32 features, and the last four convolutional layers of the decoder to 64 features.

	Time (ms)
Sample reducer	3.9
U-net	18.8
Sample partitioner	10.2
Layer filtering	5.3
Total	**38.3**

Table 3: *Inference performance at a resolution of* 1920×1080 *of the different steps of our* OPTIMIZED *architecture, measured on an NVIDIA GeForce RTX 2080 Ti. We run 250 iterations of the network and report the average timing for each pass to reduce fluctuations.*

	relMSE ↓	**SMAPE** ↓	**SSIM** ↑	**PSNR** ↑
LAYER	0.0288	0.0350	0.941	33.838
OPTIMIZED	0.0351	0.0361	0.935	33.435

Table 4: *Comparing image error metrics for the optimized versus the large layer network. Average over the 36 images in the test set.*

Our architecture contains three fully connected networks: sample reducer, sample partitioner, and kernel generator, which we in practice realize as 1×1 convolutions. A main problem with this design is that the compute-to-bandwidth ratio is relatively low, so compute kernels are typically memory bandwidth limited. A key observation is that with the hierarchical kernel prediction, all fully connected networks have low enough feature counts that we can fit weights and temporary data entirely within the GPU's shared local memory. We can optimize these networks by evaluating three consecutive fully connected layers in one compute shader without writing the data to RAM.

Runtime Performance. We use our optimized implementation to evaluate runtime performance on an NVIDIA GeForce RTX 2080 Ti GPU at a resolution of 1920×1080 using 8 spp with 20 scalar features per sample. The results are shown in Table 3. The layer filtering phase includes both the cost of the kernel generator network and the cost of applying the hierarchical filter kernel on the two layers.

Despite our efforts, performance is still limited by per-sample work (sample reducer + sample partitioning = 14 ms). Previous work [VRM*18, GLA*19, BVM*17] report GPU timings of 12-40 seconds per frame at 1080p from 8 spp, admittedly on unoptimized Tensorflow / PyTorch implementations and with a higher input feature count. We find it encouraging that it is possible to reach similar denoising quality and robustness against outliers with a two-layer architecture running at interactive rates (40 ms).

As shown in Table 4, OPTIMIZED has lower image metric scores compared to the large LAYER network. This is caused by the large reduction in network capacity, with OPTIMIZED being significantly smaller even than PIXELGATHER (see Figure 10). We note that OPTIMIZED still retains the outlier filtering and higher visual quality of our layer architecture, and is still less artifact prone than PIXELGATHER even though the numerical error is higher. Please refer to the supplemental material for image quality comparisons.

Scaling. It is beyond the scope of this paper to implement optimized versions of all networks, so we analyze scaling with respect to sample count by counting the number of fused multiply-add operations (FMAs) for PIXELGATHER, SAMPLESPLAT and LAYER. We also include OPTIMIZED, the smaller version of LAYER optimized for interactive denoising. Apart from OPTIMIZED, all algorithms run the same U-net, and the main difference in computations are at which frequency kernel generation and filtering are applied. For fairness, please focus on comparing the three large networks.

Figure 10 shows how the algorithms scale with increasing sample count. PIXELGATHER has best performance, and is invariant to sample count. Our LAYER architecture scales well, with relatively modest overhead with increasing sample count. Instinctively it feels like we should increase layer count with increasing samples, but the opposite is true. We only need enough layers to match the visibility or shading complexity of the scene. Increasing sample count reduces noise and makes the denoising problem easier.

Computational cost is a good estimate for the overall performance of a network, but note that applying a kernel predicted filter, in particular, is heavily bandwidth limited and its cost is therefore underestimated in Figure 10. This works further to our benefit, as we perform fewer filter operations than SAMPLESPLAT.

7. Conclusions

We have presented a layer-based denoising algorithm that produces image quality comparable to per-sample denoising, both visually and in image quality metrics, while being almost as efficient as denoisers working on accumulated pixel values. We denoise 1080p images at interactive rates on contemporary GPUs.

We observe similar robustness against outlier samples, a smoother look, and better handling of out-of-focus regions as first shown by Gharbi et al. [GLA*19]. In general, it seems beneficial to give the network the flexibility to apply more than one kernel per pixel. In practice, we see most benefits already with two layers.

When comparing PIXELGATHER and SAMPLESPLAT in our evaluation, the differences are fairly subtle, and smaller than what we had anticipated. This may be an effect of our reduced input feature count (20 instead of 74 floats) compared to Gharbi et al., and differences in training data and test set. In scenarios where runtime performance is critical, it remains an open question if it is worth the added cost of incorporating per-sample information in machine learning denoisers, both in terms of the additional bandwidth usage requirements and the added arithmetics of per-sample or per-layer kernels. Extreme firelies are less common in real-time rendering settings with short ray paths and smooth approximations of global illumination.

Still, we argue that a layered denoising architecture is a flexible, scalable tool to exploit per-sample data. Our architecture learns to partition samples into layers, learns unique filter kernels per layer and alpha composite the filtered layers, all trained end-to-end. We hope this research will inspire future progress in denoising for both offline and real-time rendering.

In future work, we hope to apply similar ideas to deep compositing workflows. We also want to extend the layer denoising approach to the temporal domain, by denoising sequences, similar to recent work [CKS*17, VRM*18, HMP*20]. We believe a layered representation can be beneficial to more robustly handle disocclusion and ghosting effects.

Acknowledgments

We thank Bill Dally and David Luebke for supporting the research, NVIDIA Research staff for suggestions and discussions, Jan Novák for valuable feedback, and Donald Brittain for deep learning framework code used in performance measurements. We are grateful to Gharbi et al. [GLA*19] for releasing the source code of their work.

References

[BEJM15] BAUSZAT P., EISEMANN M., JOHN S., MAGNOR M.: Sample-Based Manifold Filtering for Interactive Global Illumination and Depth of Field. *Computer Graphics Forum 34*, 1 (2015), 265–276. 2

[Bit16] BITTERLI B.: Rendering resources, 2016. https://benedikt-bitterli.me/resources/. 7

[BVM*17] BAKO S., VOGELS T., MCWILLIAMS B., MEYER M., NOVÁK J., HARVILL A., SEN P., DEROSE T., ROUSSELLE F.: Kernel-Predicting Convolutional Networks for Denoising Monte Carlo Renderings. *ACM Trans. Graph. 36*, 4 (2017). 2, 7, 9

[CFG*15] CHANG A. X., FUNKHOUSER T., GUIBAS L., HANRAHAN P., HUANG Q., LI Z., SAVARESE S., SAVVA M., SONG S., SU H., XIAO J., YI L., YU F.: *ShapeNet: An Information-Rich 3D Model Repository*. Tech. Rep. arXiv:1512.03012 [cs.GR], Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015. 5

[CKS*17] CHAITANYA C. R. A., KAPLANYAN A. S., SCHIED C., SALVI M., LEFOHN A., NOWROUZEZAHRAI D., AILA T.: Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Trans. Graph. 36*, 4 (2017), 98:1–98:12. 2, 10

[CMK*14] CIMPOI M., MAJI S., KOKKINOS I., MOHAMED S., , VEDALDI A.: Describing Textures in the Wild. In *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)* (2014). 5

[GB10] GLOROT X., BENGIO Y.: Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics* (2010), Teh Y. W., Titterington M., (Eds.), vol. 9 of *Proceedings of Machine Learning Research*, pp. 249–256. 5

[GLA*19] GHARBI M., LI T.-M., AITTALA M., LEHTINEN J., DURAND F.: Sample-based Monte Carlo Denoising Using a Kernel-splatting Network. *ACM Trans. Graph. 38*, 4 (july 2019), 125:1–125:12. 1, 2, 3, 4, 5, 7, 9, 10, 12

[GO12] GASTAL E. S. L., OLIVEIRA M. M.: Adaptive Manifolds for Real-Time High-Dimensional Filtering. *ACM TOG 31*, 4 (2012), 33:1–33:13. 2

[HMP*20] HASSELGREN J., MUNKBERG J., PATNEY A., SALVI M., LEFOHN A.: Neural Temporal Adaptive Sampling and Denoising. *Computer Graphics Forum (Proceedings of Eurographics)) 39*, 2 (2020), 147–156. 2, 9, 10

[KB15] KINGMA D. P., BA J.: Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference for Learning Representations* (2015). 5

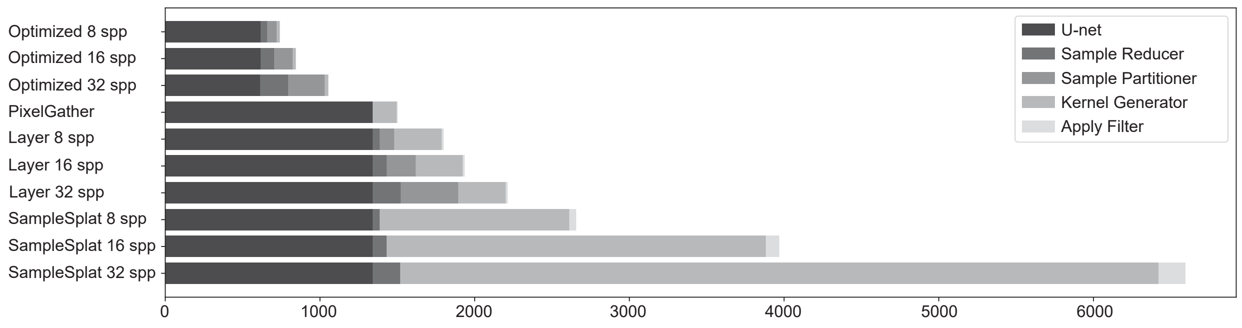

Figure 10: *Computational cost of the different algorithms at different sample counts, measured in number of FMAs (in billions). Note that we only show one result for* PIXELGATHER *since it is invariant to sample count.*

[LMH∗18] LEHTINEN J., MUNKBERG J., HASSELGREN J., LAINE S., KARRAS T., AITTALA M., AILA T.: Noise2Noise: Learning Image Restoration without Clean Data. In *Proceedings of the 35th International Conference on Machine Learning* (2018), vol. 80, pp. 2965–2974. 2

[LSR18] LEIMKÜHLER T., SEIDEL H.-P., RITSCHEL T.: Laplacian Kernel Splatting for Efficient Depth-of-field and Motion Blur Synthesis or Reconstruction. *ACM Transactions on Graphics (Proc. SIGGRAPH 2018) 37*, 4 (2018). doi:10.1145/3197517.3201379. 2

[MBC∗18] MILDENHALL B., BARRON J. T., CHEN J., SHARLET D., NG R., CARROLL R.: Burst denoising with kernel prediction networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2018). 2, 6

[McG17] MCGUIRE M.: Computer graphics archive, July 2017. URL: https://casual-effects.com/data. 7

[MVH∗14] MUNKBERG J., VAIDYANATHAN K., HASSELGREN J., CLARBERG P., AKENINE-MÖLLER T.: Layered reconstruction for defocus and motion blur. *Computer Graphics Forum (Proceedings of EGSR) 33*, 4 (2014), 81–92. 2

[NML17a] NIKLAUS S., MAI L., LIU F.: Video Frame Interpolation via Adaptive Convolution. In *CVPR* (2017). 2

[NML17b] NIKLAUS S., MAI L., LIU F.: Video Frame Interpolation via Adaptive Separable Convolution. In *ICCV* (2017). 2

[NVI18] NVIDIA: Orca: Open research content archive, 2018. https://developer.nvidia.com/orca. 7

[PGC∗17] PASZKE A., GROSS S., CHINTALA S., CHANAN G., YANG E., DEVITO Z., LIN Z., DESMAISON A., ANTIGA L., LERER A.: Automatic differentiation in PyTorch. In *NIPS-W* (2017). 5

[RFB15] RONNEBERGER O., FISCHER P., BROX T.: U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention* (2015), vol. 9351, pp. 234–241. 2, 3

[SACM96] STOKES M., ANDERSON M., CHANDRASEKAR S., MOTTA R.: A Standard Default Color Space for the Internet - sRGB, 1996. URL: https://www.w3.org/Graphics/Color/sRGB.html. 6

[Sis16] SISSON D.: Pixar One Twenty Eight, 2016. https://renderman.pixar.com/pixar-one-twenty-eight. 5

[Smi18] SMITHSONIAN: Smithsonian 3d digitization, 2018. https://3d.si.edu/. 7

[VAN∗19] VICINI D., ADLER D., NOVÁK J., ROUSSELLE F., BURLEY B.: Denoising Deep Monte Carlo Renderings. *Computer Graphics Forum 38*, 1 (2019), 316–327. 3

[VMCS15] VAIDYANATHAN K., MUNKBERG J., CLARBERG P., SALVI M.: Layered Light Field Reconstruction for Defocus Blur. *ACM Trans. Graph. 34*, 2 (2015). 2, 3, 12

[VRM∗18] VOGELS T., ROUSSELLE F., MCWILLIAMS B., RÖTHLIN G., HARVILL A., ADLER D., MEYER M., NOVÁK J.: Denoising with Kernel Prediction and Asymmetric Loss Functions. *ACM Trans. Graph. 37*, 4 (2018), 124:1–124:15. 2, 3, 6, 7, 9, 10

[WW19] WONG K.-M., WONG T.-T.: Deep Residual Learning for Denoising Monte Carlo Renderings. *Computational Visual Media 5*, 3 (2019), 239–255. 2

[XZW∗19] XU B., ZHANG J., WANG R., XU K., YANG Y.-L., LI C., TANG R.: Adversarial Monte Carlo Denoising with Conditioned Auxiliary Feature Modulation. *ACM Trans. Graph. 38*, 6 (2019), 224:1–224:12. 2

[Zaa16] ZAAL G.: HDRI Haven, 2016. https://hdrihaven.com/. 5

[ZJL∗15] ZWICKER M., JAROSZ W., LEHTINEN J., MOON B., RAMAMOORTHI R., ROUSSELLE F., SEN P., SOLER C., YOON S.-E.: Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. *Computer Graphics Forum (Proceedings of Eurographics - State of the Art Reports) 34*, 2 (2015), 667–681. 2

[ZRJ∗15] ZIMMER H., ROUSSELLE F., JAKOB W., WANG O., ADLER D., JAROSZ W., SORKINE-HORNUNG O., SORKINE-HORNUNG A.: Path-space Motion Estimation and Decomposition for Robust Animation Filtering. *Computer Graphics Forum (Proceedings of EGSR) 34*, 4 (2015). 2

Appendix A: Pseudocode: Denoiser forward pass

We show pseudocode for the forward pass of our architecture in the listing below. This code corresponds to the description in Section 3 and uses the ordered layered depth representation, based on alpha blending, as proposed by Vaidyanathan et al. [VMCS15]. A sample that hits a background layer indicates that a ray could travel unoccluded through all layers in front of it, so we increment an occupancy weight, `l_n`, for all layers in front. We apply the same kernel to radiance, alpha and occupancy, then normalize radiance and alpha with occupancy. Finally, the filtered results are composited front to back.

Note that we exponentiate filter weights on L26 to make them positive, similar to a softmax operation. When doing so, we subtract the maximum weight value to avoid numerical instabilities with large exponents [GLA*19].

```
1  def forward(samples, num_layers):
2      # Loop over samples to create embeddings
3      for i, s in enumerate(samples):
4        embedding[i] = sample_reducer(s)
5        radiance[i]  = get_linear_radiance(s)
6
7      context = UNet(mean(embedding))
8
9      # Splat samples to layers
10     for i in range(num_samples):
11       w = partition(embedding[i], context)
12       w = softmax(w) / num_samples
13       for j in range(num_layers):
14         l_radiance[j] += radiance[i] * w[j]
15         l_weight[j] += w[j]
16         l_embed[j] += embedding[i] * w[j]
17         l_n[j] += sum(w[j:num_layers])
18
19     # Compute kernel weights
20     for j in range(num_layers):
21       kernel[j] = kernel_gen(l_embed[j], context)
22
23     # Apply kernels and alpha blend
24     k = 1.0
25     for j in range(num_layers):
26       kernel[j] = exp(kernel[j] - max(kernel))
27       n     = kpn(l_n[j], kernel[j]) + eps
28       rad   = kpn(l_radiance[j], kernel[j])/n
29       alpha = kpn(l_weight[j], kernel[j])/n
30       final_color += k*rad
31       k = k*(1-alpha)
32
33     return final_color
```

Appendix B: Network Implementation Details

We implement all fully connected networks with 1×1 convolutions. All activations are leaky ReLU with a negative slope of 0.01, unless otherwise noted. The feature counts of the network stages are summarized below.

Sample reducer Three fully connected layers with input-output feature counts [input features, 32], [32, 32], and [32, 32]. In our implementation we use 20 input features and the network produces 32 features for the *sample embedding* of each sample.

Pixel U-net See Figure 3. The U-net produces 128 context features from 32 inputs (averaged sample embeddings).

Sample partitioner Three fully connected layers with feature counts [160, 32], [32, 16], and [16, layers]. No activation in the last layer. Input consists of 128 context features + 32 embedding features for a total of 160 features. The output is a scalar weight per layer.

Kernel generator Three fully connected layers with feature counts [160, 128], [128, 128], and [128, kernel $\times$ kernel]. No activation in the last layer. Input consists of 128 context features + 32 embedding features and the produced kernels have 17×17 taps.

Per-sample and Per-pixel network For the per-sample network, the sample reducer, U-net, and kernel generator are identical to the layer variant. In the per-pixel variant, we additionally compute variance over the samples and pass as input to the sample reducer, so its input feature count is doubled. Note that the kernel generator is executed once per-sample in the sample variant. The sample partitioner step appears only in the layer network.

Appendix C: Loss Function Comparison

In Table 5 we compare error metrics on the test set when training the networks with two different loss functions.

L_2 tonemap	**relMSE ↓**	**SMAPE ↓**	**SSIM ↑**	**PSNR ↑**
PIXELGATHER	0.0452	0.0371	0.932	33.427
SAMPLESPLAT	0.0358	0.0362	0.937	33.720
LAYER	0.0355	0.0361	0.937	33.668
SMAPE	**relMSE ↓**	**SMAPE ↓**	**SSIM ↑**	**PSNR ↑**
PIXELGATHER	0.0352	0.0353	0.938	33.706
SAMPLESPLAT	0.0298	0.0344	0.941	33.878
LAYER	0.0288	0.0350	0.941	33.838

Table 5: *Image quality metrics for the different algorithms when trained either using L_2 on tone mapped values (top) or SMAPE (bottom). Average over the 36 images in the test set. The relMSE and SMAPE scores are computed on linear radiance values. Lower values represent lower error. SSIM and PSNR are computed on tone mapped (Equation 7) values. Higher values means better quality. Note that all networks consistently improve on image quality metrics when trained with SMAPE.*