Random-Access Neural Compression of Material Textures

KARTHIK VAIDYANATHAN*, NVIDIA, USA
MARCO SALVI*, NVIDIA, USA
BARTLOMIEJ WRONSKI*, NVIDIA, USA
TOMAS AKENINE-MÖLLER, NVIDIA, Sweden
PONTUS EBELIN, NVIDIA, Sweden
AARON LEFOHN, NVIDIA, USA

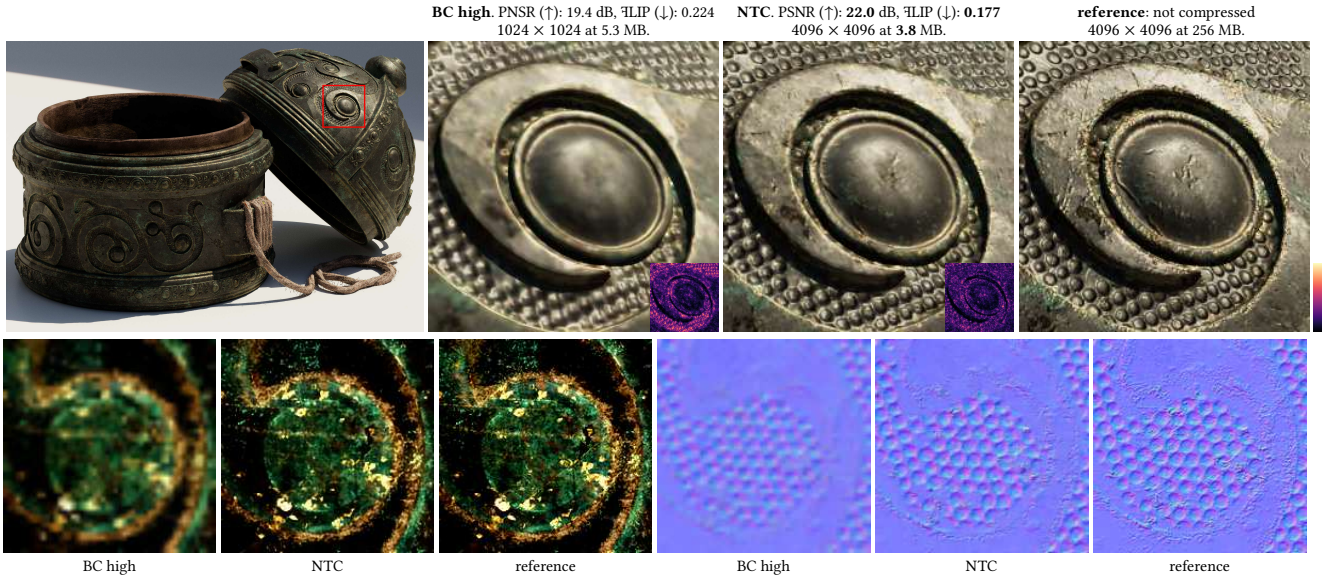

Fig. 1. A rendered image of an inkwell. The cutouts demonstrate quality using, from left to right, GPU-based texture formats (BC high) at 1024 × 1024 resolution, our neural texture compression (NTC), and high-quality reference textures. Note that NTC provides a 4× higher resolution (16× texels) than BC high, despite using 30% less memory. The PSNR and ꟻLIP quality metrics, computed for the cutouts, are shown above the respective images. The ꟻLIP error images are shown in the lower right corners, where brightness is proportional to error. Bottom row: two of the textures that were used for the renderings.

The continuous advancement of photorealism in rendering is accompanied by a growth in texture data and, consequently, increasing storage and memory demands. To address this issue, we propose a novel neural compression technique specifically designed for material textures. We unlock two more levels of detail, i.e., 16× more texels, using low bitrate compression, with image quality that is better than advanced image compression techniques, such as AVIF and JPEG XL.

At the same time, our method allows on-demand, real-time decompression with random access similar to block texture compression on GPUs, enabling compression on disk and memory. The key idea behind our approach is compressing multiple material textures and their mipmap chains together, and using a small neural network, that is optimized for each material, to decompress them. Finally, we use a custom training implementation to achieve practical compression speeds, whose performance surpasses that of general frameworks, like PyTorch, by an order of magnitude.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Image compression**; **Neural networks**.

Additional Key Words and Phrases: texture compression, neural networks

ACM Reference Format:
Karthik Vaidyanathan, Marco Salvi, Bartlomiej Wronski, Tomas Akenine-Möller, Pontus Ebelin, and Aaron Lefohn. 2023. Random-Access Neural Compression of Material Textures. 1, 1 (May 2023), 25 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Authors contributed equally to this work.

Authors' addresses: Karthik Vaidyanathan, kvaidyanatha@nvidia.com, NVIDIA, USA; Marco Salvi, msalvi@nvidia.com, NVIDIA, USA; Bartlomiej Wronski, bwronski@nvidia.com, NVIDIA, USA; Tomas Akenine-Möller, takenine@nvidia.com, NVIDIA, Sweden; Pontus Ebelin, pandersson@nvidia.com, NVIDIA, Sweden; Aaron Lefohn, alefohn@nvidia.com, NVIDIA, USA.

XXXX-XXXX/2023/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In recent years, the visual quality of real-time rendering has been approaching the levels of VFX and film productions, giving rise to powerful new workflows, like virtual production [59], which are transforming filmmaking. These improvements in quality have been achieved through the adoption of methods used in cinematic rendering such as physically-based shading for photorealistic modeling of materials [10], ray tracing [56, 70] and denoising [24, 75] for accurate global illumination, and technologies like Nanite [71] that render compressed micropolygons, thus enabling a significant increase in geometric detail.

Although there is a greater convergence of rendering techniques between cinematic and real-time applications, content creation workflows remain largely different. In order to limit storage size, games often use specialized practices for texturing models, which can require significant effort, for example, reusing content through instancing, layering tiled materials, or using procedural effects. In spite of these efforts, games typically present blurry, magnified textures close to the camera. Furthermore, some of these techniques are not applicable to uniquely parametrized content, such as photogrammetry, usage of which is a growing trend in games today. One of the main obstacles for achieving the next level of realism in real-time rendering is limited disk storage, download bandwidth, and memory size constraints.

While texture storage requirements in real-time applications have increased significantly, texture compression on GPUs has seen relatively little change. GPUs still rely on block-based texture compression methods [45, 55, 82], first introduced in the late 1990s. These methods have efficient hardware implementations and desirable properties like random access and data locality. They can achieve high, near lossless quality, but are designed for moderate compression ratios, typically between 4× and 8×. They are also limited to a maximum of 4 channels, while the number of material properties in modern real-time renderers commonly exceed this limit, thus requiring multiple textures. The main improvement to texture compression in recent years has been meta-compression for reduced disk storage and faster delivery [30], but this requires transcoding to GPU texture compression formats.

On the other hand, the field of natural image compression is making significant strides in the lower bitrate regime. In recent years, new web image formats have been proposed [2, 13] that significantly improve upon previous standards, like JPEG [80]. Meanwhile, the scientific community has been developing *neural image compression* methods [7, 14, 43], incorporating non-linear transformations in the form of neural networks, to aid compression and decompression. These methods significantly improve upon the perceptual quality of compressed images at extremely low bitrates, but typically offer modest distortion metrics improvements. They also require large-scale image data sets and expensive training, and are not suitable for real-time rendering due to their lack of important features, such as random access and non-color material properties compression.

In this work, we tackle the problem of reducing texture storage by integrating techniques from GPU texture compression as well as neural image compression and introducing a neural compression technique specifically designed for material textures.

Using this approach we enable low-bitrate compression, unlocking two additional levels of detail (or 16× more texels) with similar storage requirements as commonly used texture compression techniques. In practical terms, this allows a viewer to get very close to an object before losing significant texture detail. Our main contributions are:

- A novel approach to texture compression that exploits redundancies spatially, across mipmap levels, and across different material channels. By optimizing for reduced distortion at a low bitrate, we can compress two more levels of details in the same storage as block-compressed textures. The resulting texture quality at such aggressively low bitrates is better than or comparable to recent image compression standards like AVIF and JPEG XL, which are not designed for real-time decompression with random access.
- A novel low-cost decoder architecture that is optimized specifically for each material. This architecture enables real-time performance for random access and can be integrated into material shader functions, such as filtering, to facilitate on-demand decompression.
- A highly optimized implementation of our compressor, with fused backpropogation, enabling practical per-material optimization with resolutions up to 8192×8192 (8k). Our compressor can process a 9-channel, 4k material texture set in 1-15 minutes on an NVIDIA RTX 4090 GPU, depending on the desired quality level.

2 PREVIOUS WORK

In this section, we first review traditional texture compression (TC), techniques used in contemporary GPUs. Subsequently, we present a brief overview of natural image compression that uses entropy coding and its recent development based on deep learning. Lastly, we examine recent advances in neural rendering that are closely related to our work.

2.1 Traditional Texture Compression

Delp and Mitchell [18] introduced *block truncation coding* (BTC), which compresses gray scale images by storing two 8-bit gray scale values per 4×4 pixels and having a single bit per pixel to select one of these two gray scale values. Each pixel is stored using 2 bits per pixel (BPP). This was modified by Campell et al. [11] who used the 8-bit values as indices into a lookup table of colors, enabling color image compression at 2 BPP. Their method is called color cell compression (CCC). Knittel et al. [35] described hardware for decompressing CCC textures, which was selected due its random-access nature and simplicity, which made it affordable and fast in hardware.

The S3 texture compression (S3TC) schemes [82] are clever extensions of the BTC and CCC and form the basis for most of the TC methods found in DirectX [45]. The first method of S3TC, which was later called DXT1 and then renamed to BC1 in DirectX, stores two colors per 4×4 pixels. These are quantized to $5 + 6 + 5$ (RGB) bits. Two additional colors are created using linear interpolation between the stored colors. Hence, there is a palette of four colors available per 4×4 pixels and each pixel then points to one of these using a 2-bit index.

Today, there are seven variants of S3TC in DirectX. These are called BC1-BC7 and handle alpha, normal maps, high-dynamic range (HDR), and are using either 4 or 8 BPP. All of these have the random access property, since each block of 4×4 pixels always are compressed to the same number of bits.

Munkberg et al. [49] and Roimela et al. [62] presented the first TC schemes for HDR textures, and both were inspired by the previous block-based schemes, but adapted those to HDR. BC6H is a variant for HDR texture compression in DirectX. We omit many other references on this topic, since HDR TC is not our focus.

Fenney [20] presented a different block-based compression scheme called PowerVR texture compression (PVRTC), which is used on all iOS devices. PVRTC decompresses two low-resolution images, which are bilinearly magnified, and then uses a per-pixel index to select a color in between the interpolated colors.

Ericsson texture compression (ETC1), which is part of OpenGL ES, also compresses 4×4 pixels at a time but stores only a single base color, which is then modulated using a trained table of offsets, which is selected per block [66]. ETC2 is backwards compatible with ETC1 by using invalid bit combinations, and improves image quality [67]. ETC1/ETC2 are available in over 12 billion mobile phones. ASTC [55] is currently the most flexible texture compression scheme, since it supports low-dynamic range, HDR, and 3D textures, with bitrates from 0.89 to 8 BPP. This is achieved using larger block sizes, specific color spaces, and efficient bit allocation. ASTC is supported on (at least) ARM's GPUs, the most recent Apple GPUs, as well as several desktop GPUs. However, there is no support in DirectX.

2.2 Traditional and Neural Image Compression

Image compression formats that target storage on disk or network transfer have less restrictive constraints than GPU texture compression. Without the need for random access and strict bounds on hardware complexity, they can utilize global transforms, as well as entropy coding methods to target significantly lower bitrates.

Despite its widespread usage, JPEG [80] has been found to produce noticeable artifacts, such as detail loss, discoloration, and banding, particularly at lower bitrates. This has led to the development of alternative image compression formats, such as AVIF [13] and JPEG XL [2], which incorporate algorithmic advancements and prioritize alignment with human visual perception [1].

The tradeoff between objective distortion and perceptual measurements has been well-studied [8], particularly in the context of machine learning. Various methods have been proposed to improve optimization for perceptual qualities, such as using features extracted from a convolutional network [88], the network structure itself [77], or another network [37]. Non-neural methods have also been developed to localize perceptually-relevant errors for image comparison [3].

In recent years, neural image compression methods [5, 61, 73] have emerged as an alternative to traditional formats such as JPEG 2000 [65], offering improved perceptual quality. These techniques often use encoder-decoder architectures to create an information bottleneck, which is then quantized and entropy-coded based on an entropy model. Rapid evolution has occurred in this area, with advances such as the provision of sideband information to improve the accuracy of the entropy model [7], the incorporation of generative models to achieve higher perceptual quality [43], and more recently the use of attention based networks [14, 40] which were the first to improve upon both PSNR and perceptual quality over the new VVC-intra [9] standard, which uses traditional compression methods.

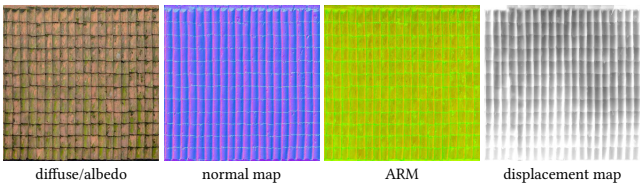

Fig. 2. An example *texture set* consisting of a diffuse map, normal map, an ARM (ambient occlusion, roughness, metalness) texture, and a displacement map, for a ceramic roof material. Our approach compresses these textures together. Textures retrieved from https://polyhaven.com/.

2.3 Neural Rendering and Materials

Neural rendering [72] is a new field that has emerged recently and includes approaches that leverage neural methods inside traditional rasterization or ray tracing based renderers, which is particularly relevant to our work. Thies et al. [74] proposed a method for higher quality image synthesis from low quality 3D content, by storing learned neural features in a texture, which are then sampled and rasterized into an off-screen buffer. The final images are then produced using a jointly-optimized neural renderer.

The idea of using neural latent grids to store spatially-varying appearance has been adopted by computer vision research, as well as computer graphics, where it has been used to represent complex materials and their mipmap chains [36]. Most of these works focus on modeling materials and their appearance, using representations that can be significantly larger than traditional 8-bit textures. While our study focuses on practicality and cost-efficiency in traditional rendering, it is important to note that our algorithm is also applicable to neural rendering, where it could greatly reduce memory consumption.

Neural networks have emerged as a popular alternative to discrete grids for signal representation. The predominant architectures are coordinate networks [46], which offer a fully differentiable and smooth representation, advantageous for 3D computer vision reconstruction tasks. Coordinate networks frequently employ *positional encoding*, a concept originating from language modeling literature [79]. Instead of passing the input coordinates $\mathbf{p}$ directly to the MLP, this method encodes it as a vector of $\sin(2^h \pi \mathbf{p})$ and $\cos(2^h \pi \mathbf{p})$ terms, where h represents an octave. Fourier encoding has been shown to overcome the low-frequency bias of MLPs [69]. For improved computational efficiency, trigonometric functions can be replaced by triangle waves [48]. As an alternative to positional encoding, trigonometric [64], Gaussian [15], or wavelet [63] activation functions can be used to increase the bandwidth of each layer. This property can be used to band limit network parts and interactively stream only lower frequencies of the encoded content [38].

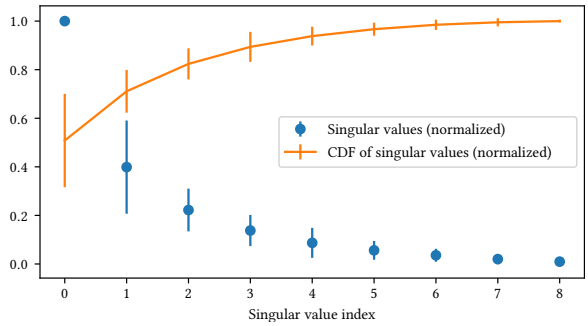

Fig. 3. Singular value distribution representing cross-channel correlation in 20 texture sets selected from diverse materials. The sharp falloff in the singular values indicates a high degree of correlation across channels.

Storage size is often overlooked in neural representation literature, with typical coordinate networks requiring more storage to represent discrete 2D signals than their uncompressed form. For instance, on the task of image fitting, the original work on positional-encoded coordinate networks [69] uses a network with 327K parameters to represent 197K scalar values (a 256×256 RGB image). Similarly, work on periodic activation functions [64] uses 329K parameters to represent 786K scalar values (a 512×512 RGB image). For this representation size, both approaches report only a PSNR under 30 dB. Improving the storage efficiency of these coordinate-based networks was a key motivation for subsequent work that used grid-based neural representations compressed using vector quantization [68] or hash tables [47].

Finally, previous work on neural radiance caches [48] has shown that it is possible to efficiently embed small neural networks inside a renderer, enabling training and inference in real time, orders of magnitude faster than traditional deep learning frameworks. We build on this work and optimize it further.

3 MOTIVATION

Lossy image compression techniques typically exploit both spatial and cross-channel correlations and, subsequently, quantize or eliminate weakly correlated information. For example, the BC1 compression format maps all RGB color triplets in a 4×4 texel tile to a single line in RGB space, assuming perfect correlation between all channels. Other BCx formats use similar assumptions but relax some constraints, such as the allowed number of lines inside a block. Block compression formats, however, can only compress textures with up to four channels, while modern renderers typically use several material properties, including diffuse color, normals maps, height maps, ambient occlusion, glossiness, roughness, and other BRDF information. These properties are typically stored as multiple textures within a group, which we refer to as a *texture set* (Figure 2). As seen in Figure 3, there is significant correlation across the channels of different textures in a texture set. This can be attributed to both the physical properties of real-world materials (specularity and albedo are inversely correlated), geometric properties (displacement, normal maps, edges), as well as the material authoring process, where an artist may layer, mask, and combine multiple channels together [51].

Earlier work [84, 85] has noted this correlation, applying it to dimensionality reduction of material inputs. Besides correlations across pixels and channels, Zontak et al. [89] have also noted redundancies across multiple scales. In this paper, we derive a neural compression scheme that builds on these observations and exploits redundancies spatially, across mip levels, and between all channels of a texture set.

4 NEURAL MATERIAL TEXTURE COMPRESSION

We represent the texture set as a tensor with dimensions $w \times h \times c$ and our model compresses the tensor without making any assumptions about the channel count or the specific semantics of each channel. For example, the normals or diffuse albedo could be mapped to any channels without affecting compression. This is possible because we learn the compressed representation for each material individually, effectively specializing it for its unique semantics. The only assumption we make is that each texture in a texture set has the same width and height. Some materials can have BRDF properties not present in other ones, for instance, subsurface scattering color or thickness. While an alternative approach using a pre-trained global encoder could potentially achieve faster compression, it would also require imposing globally pre-assigned semantics for each channel, which can be impractical for a large set of diverse materials.

Figure 4 illustrates the decoding process, progressing from a compressed representation, on the left, to a decompressed texel, on the right. Our compressed representation is a pyramid of quantized features levels, which are typically at a lower resolution compared to the reference texture. To achieve decompression of a single texel, feature vectors are sampled from a feature level and subsequently decoded to generate all channels within the texture set. To facilitate greater feature decorrelation, the decoder is modeled as a non-linear transform [6], utilizing a multilayer perceptron (MLP) as a universal approximator [28]. This MLP is shared across all the mipmap (mip) levels, which enables joint learning of the compressed representation and the MLP's weights, using an autodecoder framework [57]. Specifically, the compressed representation is directly optimized through quantization-aware training and backpropagation through the decoder, as opposed to using an encoder.

The following sections describe each stage of decompression in detail. Later, in Section 6, we show how those assumptions hold over a diverse set of materials and textures from different datasets, different formats, and using different material semantics.

4.1 Feature Pyramid

As shown in Figure 4 (a), our compressed representation is a pyramid of multiple *feature levels* F^j, with each level, j, comprising a pair of 2D *grids*, G_0^j and G_1^j. The grids' cells store *feature vectors* of quantized latent values, which are utilized to predict multiple mip levels. This sharing of features across two or more mip levels lowers the storage cost of a traditional mipmap chain from 33% to $\sim$ 6.7% or less. Furthermore, within a feature level, grid G_0 is at a higher resolution, which helps preserve high-frequency details, while G_1 is at a lower resolution, improving the reconstruction of low- frequency content, such as smooth gradients.

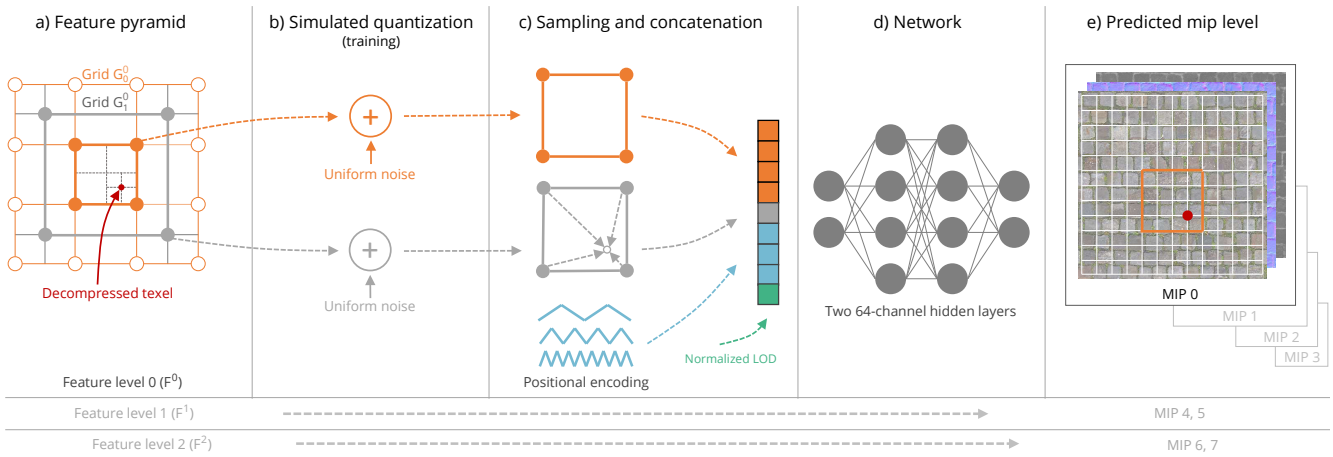

Fig. 4. Overview of our method. a) Our compressed representation comprises multiple feature levels, each having two feature grids; a high resolution grid G_0 and a low resolution grid G_1 (Section 4.1). The solid circles represent the grid cells accessed for a target texel (in red). b) During training, we simulate quantization through addition of noise and clipping (Section 4.2). c) During inference and training, we sample the four neighboring feature vectors (orange circles) from the grid G_0 (Section 4.3.1) and bilinearly interpolate features from G_1 (hollow gray circle), concatenating them with local positional encoding (Section 4.3.2) and a normalized level-of-detail (LOD) value for the target mip level. d) Finally, we use a neural network (Section 4.4) to decode the mip level (e).

Table 1 illustrates the feature levels and grid resolutions for a 1024×1024 texture set. The resolution of the grids is significantly lower than the texture resolution, resulting in a highly compressed representation of the entire mip chain. Typically, a feature level represents two mip levels, with some exceptions; the first feature level must represent all higher resolution mips (levels 0 to 3), and the last feature level represents the bottom three mip levels, as it cannot be further downsampled.

4.2 Simulated Quantization

Since we do not use entropy coding, we enforce a fixed quantization rate for all latent values in a feature grid and only optimize for image distortion. We simulate quantization errors along the lines of previous neural image compression techniques [5] by adding uniform noise in the range $\left(-\frac{Q_k}{2}, \frac{Q_k}{2}\right)$ to the features, where Q_k is the range of a quantization bin on grid k. To limit the number of quantization levels, we clamp features to the quantization range after updating them in the backward pass. This ensures that both gradient computations and feature updates are w.r.t. values strictly within the quantization range. We observed that this approach produces better results than clamping features in the forward pass, where the learned features can drift outside the desired quantization range.

Table 1. Compressed representation of a mip chain through feature levels and low resolution grids for a 1024×1024 texture set.

Feature level F^j	G_0^j grid resolution	G_1^j grid resolution	Predicted mip levels
0	256×256	128×128	0,1,2,3
1	64×64	32×32	4,5
2	16×16	8×8	6,7
3	4×4	2×2	8,9,10

For each feature grid G_k^j, we use an asymmetric quantization range of $\left[-\frac{N_k-1}{2}Q_k, \frac{N_k}{2}Q_k\right]$, where $N_k = 2^{B_k}$ is the desired number of quantization levels. This quantizes a zero value with no errror by aligning it with the center of a quantization bin [32]. In turn, this produces better results especially when we quantize to four levels or less. We set Q_k to $\frac{1}{N_k}$ and therefore N_k is the only value provided during training. Toward the end of the training process, we stop adding noise to simulate quantization and explicitly quantize the feature values. The feature values are frozen for the rest of the training. Then, we continue to optimize the network weights for 5% more steps, adapting them to the discrete-valued grids. We also include a comparison between scalar quantization and vector quantization [78] in our supplementary material (Appendix E).

4.3 Sampling and Concatenation

In this section, we describe the first stage of decompression, which samples the grids of a feature level and prepares the input to the MLP, as shown in Figure 4 (c). In this stage, we first select a feature level based on the desired level of detail (LOD) (Table 1), and then resample both the grids in the feature level to the target resolution. In the next section, we describe how grids are resampled by interpolating the features at the target texel location. Following this, we describe our positional encoding scheme that aids in interpolation and preserving high-frequency details.

4.3.1 Feature Interpolation. Features may be upsampled or downsampled depending on the feature level and the target LOD. However, upsampling the first feature level F^0 alone presents the main challenge for reconstruction quality, as it is typically at a much lower resolution than the input texture. To a large extent, we rely on the lower resolution of the grids for compression.

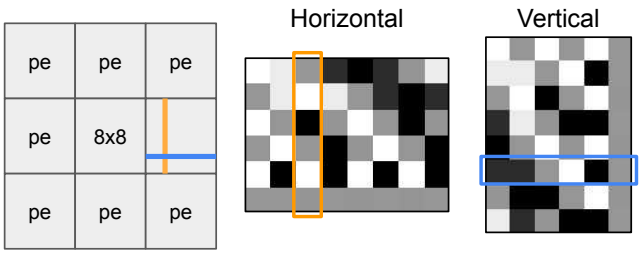

Fig. 5. Positional encoding tiles of 8 × 8 texels. A single texel is represented by 6+6 scalars, each encoding the horizontal and vertical texel position inside the tile. The last value is constant in both the horizontal and vertical encoding.

To achieve real-time decompression performance, we balance complexity against reconstruction quality by using two different approaches for resampling the grids. We use a *learned interpolation* approach for the higher resolution grid G_0 and bilinear interpolation for the lower resolution grid G_1. In the case of learned interpolation, we concatenate four neighboring feature vectors and rely on phase information from the positional encoding (Section 4.3.2) to reconstruct high-frequency details. Concatenation, as opposed to summation of weighted features, allows the following MLP layers to combine features differently depending on the texel location. However, the learned interpolation also increases the cost of the input layer of the network. The bilinear interpolation of the low resolution grid was chosen to limit this. We observed that the smooth output of bilinear interpolation can compliment learned interpolation well by suppressing banding artifacts resulting from heavily quantized features.

4.3.2 Tiled Positional Encoding. To improve the fidelity of high-frequency details, we condition our decoder on *positional encoding* [46, 79]. We use a more computationally efficient variant of the encoding [48], which is based on triangular waves, and observe no quality loss.

Our architecture is not fully coordinate-based since we also use features stored in low-resolution grids. Therefore, any low-frequency information can be directly represented by the features and we only need positional encoding to represent frequencies higher than the Nyquist limit of the grids. The number of octaves for the encoding is $\log_2 8$, as 8 is the maximum upsampling factor we encounter, i.e., when upsampling grid G_1 to a target LOD of 0. Consequently, the encoding is a tiled pattern that repeats every 8 × 8 texels, as shown in Figure 5.

Fig. 6. With a low-bitrate model, the choice of loss function can improve color fidelity or preservation of high-frequency details but typically not both. **Left:** Original. **Middle:** L2. **Right:** Linear combination of L2 and 1 − SSIM. Textures retrieved from https://ambientcg.com.

4.4 Network

Our network is a simple multi-layer perceptron with two hidden layers, each of size 64 channels. The size of our input is given by $4C_0 + C_1 + 12 + 1$, where C_k is the size of the feature vector in grid G_k. Note that we use 4× more features from grid G_0 for learned interpolation, 12 values of positional encoding and a LOD value.

We do not use any activation function on the output of the last layer. We experimeted with several different activation functions for the three remaining layers and observed best results with GELU [26]. To reduce the computation overhead of GELU functions, we derived an approximation denoted "hardGELU", which is similar to hard Swish [29]. Our variant is given by

$$\text{hardGELU}(x) = \begin{cases} 0, & \text{if } x < -\frac{3}{2}, \\ x, & \text{if } x > \frac{3}{2}, \\ \frac{x}{3}(x + \frac{3}{2}), & \text{otherwise.} \end{cases}$$

4.5 Optimization Procedure and Loss Function

We jointly optimize the feature pyramid and the decoder, using gradient descent with the ADAM [34] optimizer. Unless stated otherwise, our model is trained for 250k iterations. Our method can use and minimize an arbitrary image loss function.

To optimize our compressed representation, we explored several different loss functions, including SSIM [81], a version of VGG loss that supports texture sets [12], adversarial as well as L1 and L2 losses, and combinations thereof. In general, we found that the loss function presented a compromise between maintaining color fidelity and preserving high-frequency details – though we were unable to find a loss function that did not show weaknesses in one or the other. Figure 6 illustrates this behavior, where using only L2 results in loss of high-frequency detail, while adding SSIM improves on this but discolors the image. The choice of objective function can thus be adapted based on the use case and when the application only requires one of the two quality properties to be preserved. We found the L2 loss to be a reasonable compromise. As it also trains robustly and is the simplest and computationally fastest choice, we use it throughout this paper.

We hypothesize that the observed behavior is a consequence of information theoretical limitations, i.e., that we cannot preserve both high-frequency detail and color fidelity at this low bitrate. Further investigation of this hypothesis is left for future work. In addition, we conducted initial experiments to explore potential benefits of using different specialized objective functions for different texture types. While these experiments did not indicate advantages of such an approach, we believe it to be an interesting area of future research.

5 IMPLEMENTATION

As outlined in in Section 4, we decompress textures at a given texel by sampling the corresponding latent values from a feature pyramid and decoding them using a small MLP network. Our compressed representation, as mentioned previously, is trained specifically for each texture set. Specializing the compressed representation for each material allows for using smaller decoder networks , resulting in fast optimization (compression) and real-time decompression.

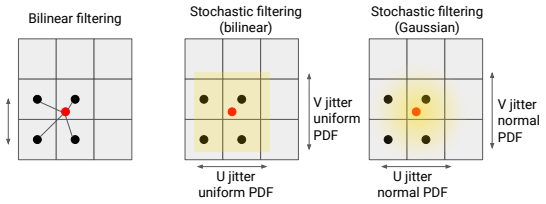

Fig. 7. Bilinear and stochastic filtering.

5.1 Compression

Similar to the approach used by Müller et al. for training autodecoders [47], we achieve practical compression speeds by using half-precision tensor core operations in a custom optimization program written in CUDA. We fuse all of the network layers in a single kernel, together with feature grids sampling, loss computations, and the entire backward pass. This allows us to store all network activations in registers, thus eliminating writes to shared or off-chip memory for intermediate data.

We process batches of eight randomly sampled 256×256 texel crops, selected from the same level of detail. For each batch, we randomly choose a level of detail proportionally to the mip level's area by sampling from an exponential distribution: $\text{LOD} = \lfloor -\log_4 X \rfloor$, where $X \sim U(0, 1)$. To mitigate undersampling of low-resolution mip levels, 5% of the batches sample their LOD from a uniform distribution defined over entire range of the mip chain. We use a high initial learning rate of 0.01 for the latent grids, and a lower value of 0.005 for the network weights and apply cosine annealing [42], lowering the learning rate to 0 at the end of training.

5.2 Decompression

Inlining the network with the material shader presents a few challenges as matrix-multiplication hardware such as tensor cores operate in a SIMD-cooperative manner, where the matrix storage is interleaved across the SIMD lanes [54, 86]. Typically, network inputs are copied into a matrix by writing them to group-shared memory and then loading them into registers using specialized matrix load intrinsics. However, access to shared memory is not available inside ray tracing shaders. Therefore, we interleave the network inputs *in-registers* using SIMD-wide shuffle intrinsics.

We used the Slang shading language [25] to implement our fused shader along with a modified Direct3D [44] compiler to generate NVVM [52] calls for matrix operations and shuffle intrinsics, which are currently not supported by Direct3D. These intrinsics are instead directly processed by the GPU driver. Although our implementation is based on Direct3D, it can be reproduced in Vulkan [23] without any compiler modifications, where accelerated matrix operations and SIMD-wide shuffles are supported through public vendor extensions. The NV_cooperative_matrix extension [22] provides access to matrix elements assigned to each SIMD lane. The mapping of these per-lane elements to the rows and columns of a matrix for NVIDIA tensor cores is described in the PTX ISA [54]. The KHR_shader_subgroup extension [21] enables shuffling of values across SIMD lanes in order to assign user variables to the rows and columns of the matrix and vice versa. These extensions are not restricted to any shader types, including ray tracing shaders.

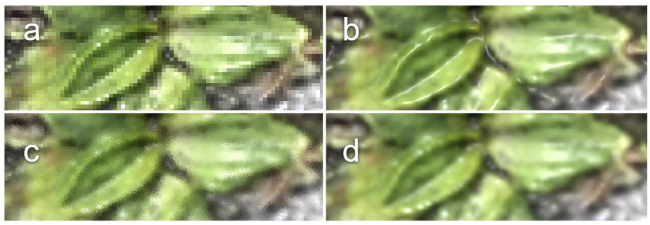

Fig. 8. Filtering across the boundaries of a highly specular material. **a)** Nearest-neighbor filtering. **b)** Trilinear filtering. **c)** Single frame of stochastic filtering. **d)** Resolved stochastic temporal filtering. Trilinear texture filtering causes specular lighting artifacts as the high specularity interpolates outside the glossy material. Textures retrieved from https://ambientcg.com.

5.2.1 SIMD Divergence. In this work, we have only evaluated performance for scenes with a single compressed texture-set. However, SIMD divergence presents a challenge as matrix acceleration requires uniform network weights across all SIMD lanes. This cannot be guaranteed since we use a separately trained network for each material texture-set. For example, rays corresponding to different SIMD lanes may intersect different materials.

In such scenarios, matrix acceleration can be enabled by iterating the network evaluation over all unique texture-sets in a SIMD group. The pseudocode in Appendix A describes divergence handling. SIMD divergence can significantly impact performance and techniques like SER [53] and TSU [31] might be needed to improve SIMD occupancy. A programming model and compiler for inline networks that abstracts away the complexity of divergence handling remains an interesting problem and we leave this for future work.

5.3 Filtering

Our method supports mipmapping for discrete levels of minification (Section 4.1), similar to BCx compression methods. For the best quality and compression ratios, our compression approach relies on overfitting the network only at the discrete texel locations in the original texture. However, this does not guarantee smooth reconstruction in between these discrete texel locations and mipmap levels. Therefore, we cannot rely on hardware acceleration for trilinear filtering and we implement it in software on the GPU. The software implementation decompresses and combines four texels for bilinear filtering, and eight texels for trilinear filtering, significantly increasing decompression cost.

In order to decouple the decompression cost from filtering, we propose a simple alternative to trilinear filtering based on stochastic sampling [17, 19, 27], which we call *stochastic filtering*. We add random noise to the (u, v) position, followed by nearest neighbor sampling. We can achieve different types of texture filtering by changing the distribution of the noise, as shown in Figure 7. For example, a uniform distribution in the range $(-0.5, 0.5)$ of one texel produces bilinear filtering, and a normal distribution produces Gaussian filtering. In addition to jittering the (u, v) coordinates, we also jitter the LOD to enable a smooth transition between mip levels.

Stochastic filtering typically increases the amount of noise in the rendered image, but we observed that modern post-process reconstruction techniques [4, 33, 39] can effectively suppress this noise.

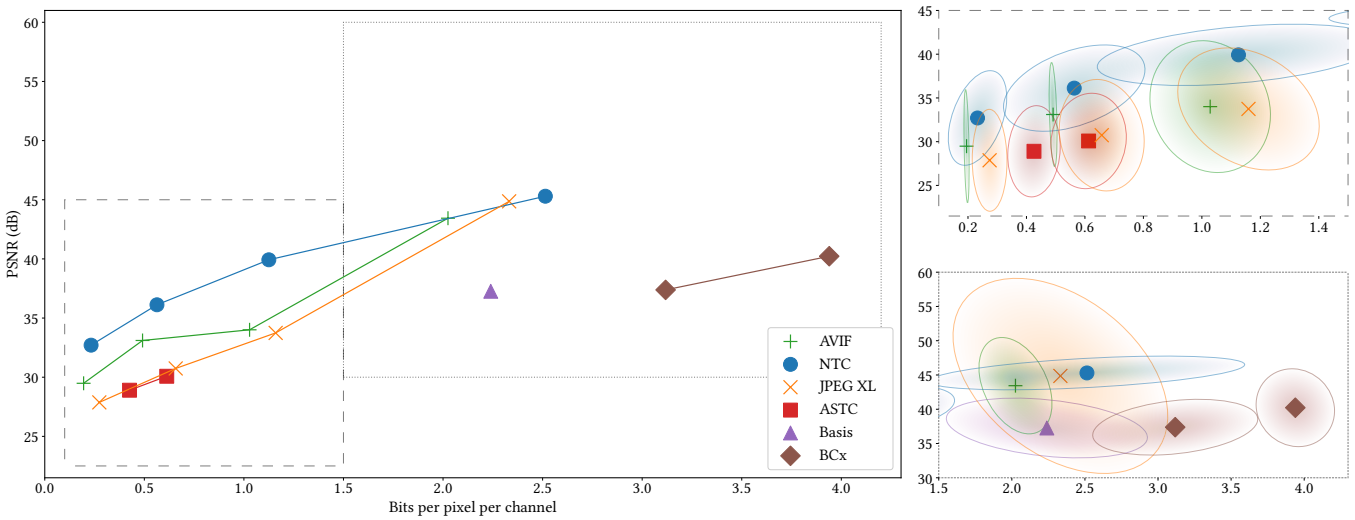

Fig. 9. Quantitative results. Vertical axis: PSNR scores of different methods. The horizontal axis is bits per pixel per channel storage cost. The size of each ellipse corresponds to the variation in PSNR and compressed material size due to different texture channel counts present in the data set. Table 3 contains the explicit numbers for the markers in this figure, in addition to corresponding SSIM and LPIPS values.

Figure 8 shows a comparison of trilinear filtering and stochastic filtering with reconstruction using DLSS [39]. We note that, while traditional texture filtering filters the input material properties, stochastic filtering filters the shading output, which produces more accurate results, as shown in Figure 8.

6 RESULTS

Our neural compression method for material textures (NTC) is flexible, allowing for adjustments in feature grids resolution, channel counts, and bit depths to optimize the trade-off between compression quality, storage, and inference performance. We use the four compression profiles listed in Table 2, each targeting a different number of bits per-pixel per-channel (BPPC), for our comparisons. Table 2 also shows the total storage cost including network weights and feature levels. The cost of the network weights is roughly constant across all profiles, and does not change with the texture resolution. Please refer to Appendix F in the supplementary material for more details of the NTC storage cost.

6.1 Evaluation Data Set

To evaluate different compression techniques, we selected 20 diverse materials with texture sets varying in content, frequency characteristics, resolution, and channel counts. The content includes natural textures, human-made objects, character textures, and a synthetic gradient, with attributes such as high-frequency repeating patterns, noisy, and smooth. The texture sets have resolutions varying from 2048×2048 to 8192×8192 texels and channel counts ranging from 3 to 12. More details about the evaluation data set can be found in our supplementary material (Appendix J).

6.2 Compared Methods

Our method can replace GPU texture compression techniques, such as BC [45] and ASTC [55]. It is a common industry practice to use different BC variants for different material texture types [16], but there is no single standard. As such, we propose two compression profiles for the evaluation of BC, namely "BC medium" and "BC high." The BC medium profile uses BC1 for diffuse and other packed multi-channel textures, BC7 for normals, and BC4 for any remaining single-channel textures. The BC high profile, on the other hand, uses BC7 for three-channel textures and BC4 for one-channel textures.

Our method is not directly comparable with compression formats using entropy encoding, as NTC is designed to support real-time random access. However, to provide a frame of reference for storage and quality at lower bitrates, we also evaluate methods using entropy encoding, namely, the texture meta-compression algorithm Basis Universal [30] and the recently standardized, high-quality image compression formats AVIF and JPEG XL. We have not included recent neural image compression techniques in our analysis as they do not produce significantly better quantitative results compared to traditional methods on metrics like MSE, despite typically offering better perceptual characteristics [14]. In addition, they do not support random access, which is a key property for GPU texture accesses. Details of the compression settings used for our evaluation are included in our supplementary material (Appendix D).

Table 2. NTC profiles with different bits per-pixel per-channel (BPPC) and total storage costs calculated using a $4096 \times 4096 \times 9$ texture set as reference

BPPC Profile	G_0^0 grid resolution	G_0^j grid channels	G_1^j grid channels	Total (MB)
NTC 0.2	1024×1024	8×2b	12×4b	3.52
NTC 0.5	1024×1024	12×4b	20×4b	8.53
NTC 1.0	2048×2048	12×2b	10×4b	17.03
NTC 2.25	2048×2048	16×4b	12×4b	38.03

Table 3. Average PSNR, 1 − SSIM, and LPIPS values over the evaluation data set for the methods used in our comparison. The methods are grouped based on their storage requirements. Algorithms marked in gray do not support random access. "BC M." is short for "BC medium" and "BC H." is for "BC high."

	Low (~ 0.2 BPPC)			Medium-low (~ 0.5 BPPC)					Medium (~ 1.0 BPPC)			Medium-high (~ 1.5 - 3.0 BPPC)					High (~ 4.0 BPPC)
	AVIF	NTC	JPEG XL	ASTC 12 × 12	AVIF	NTC	ASTC 10 × 10	JPEG XL	AVIF	NTC	JPEG XL	AVIF	Basis	JPEG XL	NTC	BC M.	BC H.
BPPC (mean)	0.20	0.23	0.27	0.43	0.49	0.56	0.61	0.66	1.03	1.13	1.16	2.02	2.24	2.33	2.51	3.12	3.94
PSNR (↑)	29.49	**32.71**	27.87	28.90	33.10	**36.12**	30.08	30.74	34.01	**39.92**	33.74	43.44	37.27	44.86	**45.30**	37.38	40.23
1 - SSIM (↓)	0.1138	**0.0633**	0.1377	0.1034	0.0586	**0.0375**	0.0788	0.0778	0.0380	**0.0183**	0.0451	0.0068	0.0219	**0.0030**	0.0076	0.0211	0.0099
LPIPS (↓)	0.1051	**0.0660**	0.1001	0.0722	0.0302	0.0364	0.0460	**0.0272**	0.0108	0.0176	**0.0087**	0.0013	0.0114	**0.0003**	0.0057	0.0133	0.0024

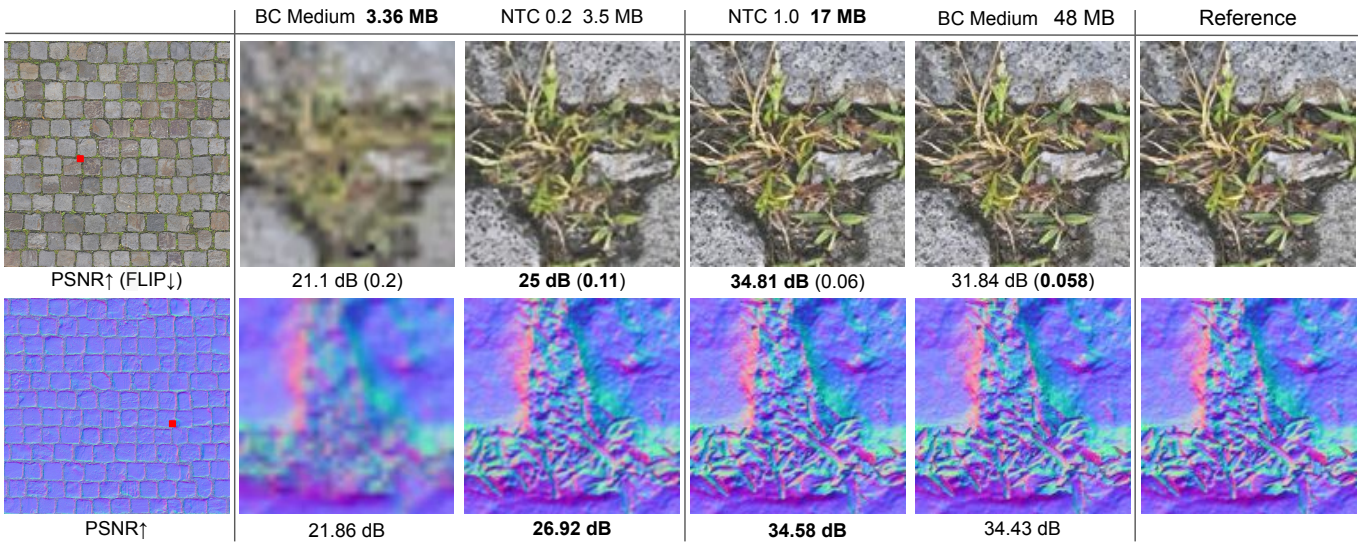

Fig. 10. Iso-storage and iso-quality comparison showing PSNR and ꟻLIP scores for the diffuse and normal map textures in the *Paving Stones* texture set, retrieved from https://ambientcg.com. For iso-storage comparisons, we use two higher mip levels for the BCx textures to match the storage size of NTC.

6.3 Quantitative Results

Figure 9 presents an overview of our compression results, showing PSNR values for different compression methods, across a range of BPPC rates as well as the variations across texture sets. The PSNR values are computed over the entire texture set and all mip levels down to resolution 4×4 (see Appendices I and J in our supplementary material). The plot shows that our method significantly outperforms GPU texture compression at high bitrates and surpasses advanced compression methods at lower rates of less than 1.5 BPPC. We attribute these improvements to significant cross-channel and cross-mip-level correlations, not exploited in prior works.

Our compression method can significantly reduce texture sizes, which can be leveraged in different ways. For example, the NTC 0.5 profile can be used to achieve iso-quality results as the BC medium profile, at 1/5th of the storage cost. Alternatively, the NTC 0.2 profile can be used to store two additional higher mip levels at a PSNR that is slightly lower but still better quality than AVIF and JPEG XL. This enables significantly higher detail, as demonstrated in Figure 1.

Although our compression is not optimized for perceptual quality, we also include SSIM [81] and LPIPS [88] metrics in Table 3 for comparison. Even with these perceptual metrics, NTC provides better results than all other compression methods at low and medium-low rates, only trailing recent high-quality image compression techniques at higher rates. Perceptual quality of our method can be further improved by optimizing for perceptual metrics.

The quantitative results with our compression technique are also consistent across different mip levels and different material textures. We include per-mip and per-texture-type PSNR results in our supplementary material (Appendix C).

6.4 Qualitative Results

Previous work has noted that the PSNR metric is not sufficient for image quality comparison and, furthermore, that *objective* distortion metrics are at an inherent trade-off with perceptual quality [8]. Unfortunately, there is no single metric that would perfectly correlate with human preferences, which might vary for different applications. We use qualitative, human analysis to evaluate image quality in addition to the PSNR metric. We also include ꟻLIP [3] values, which are better aligned with perceptual quality.

6.4.1 Texture Quality. A key motivation for our work was to determine the extent to which we could preserve image quality while reducing the storage. Figure 10 provides an overview of this by comparing the BC medium profile with NTC at different rates. We present an approximately *iso-quality* comparison using the medium rate NTC 1.0 profile and *iso-storage* comparison using the low rate using NTC 0.2 profile. To evaluate BC medium at a low BPPC rate, we excluded the two largest mip levels to achieve a comparable storage size to NTC 0.2. For both comparisons, we used the 4096×4096 resolution *Paving Stones* texture set, which is one of the most challenging in our evaluation.

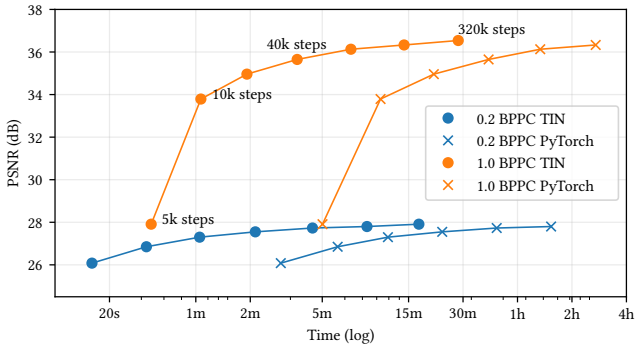

Fig. 11. PSNR vs. training time for a 4096×4096 texture set with 9 channels. Our custom training implementation is an order of magnitude faster than PyTorch, compressing the texture set in a few minutes.

In the iso-quality comparison, we see that the BC medium profile consumes 48 MB of storage, while the NTC 1.0 profile exceeds its quality with just 1/3rd of the size. We also see that the NTC 0.2 profile only consumes 3.5 MB of storage and has a significantly higher quality that appears closer to the reference than the BC medium profile at a comparable size.

Figure 13 presents a more extensive iso-storage comparison for the NTC 0.2. We selected the 8192 × 8192, *painted concrete* texture set for this comparison as it is close to the mean PSNR score reported in Table 3. Overall, our method produces results that are significantly better than BCx compression with the same storage. We also observe better normal map quality compared to advanced image compression techniques like AVIF and JPEG XL, while the diffuse texture is slightly blurrier. We provide a more detailed qualitative analysis of this texture set in our supplementary material (Appendix D) and cover a set of failure cases in Section 7.2.

6.4.2 Rendered Quality. Since our method is designed for compressing material textures used in rendering, we demonstrate the results with end-to-end rendered images in Figure 1. The figure compares the NTC 0.2 profile, which uses 3.8 MB for the metal texture set, to "BC high" with two mip levels removed and still using 39% more storage. As depicted in the insets, the quality of the NTC 0.2 profile is superior, which is also indicated by the better PSNR and ꟻLIP numbers. Our supplementary material includes additional examples, including a closed book and an open book with text, to further demonstrate the effectiveness of our method.

6.4.3 Filtering Quality. In the supplementary video, we showcase the quality of stochastic temporal filtering in motion. We use DLSS for high-quality spatiotemporal reconstruction [39] as described in Section 5.3. The use of temporal reconstruction techniques may exhibit flickering or ghosting in certain scenarios. Our experiments reveal minor specular flickering, but no ghosting, under fast motion. A higher quality jitter sequence [83], or reconstruction techniques optimized for stochastic filtering, could further improve quality.

Table 4. Decompression performance for a 4k material texture set (*Paving Stones*). Performance is similar across all texture sets for a given profile.

BC High	NTC 0.2	NTC 0.5	NTC 1.0	NTC 2.25
0.49 ms	1.15 ms	1.46 ms	1.33 ms	1.92 ms

6.5 Performance

In this section, we discuss compression performance as well as decompression performance in a simple renderer.

6.5.1 Compression. As described in Section 5.1, we use a custom CUDA-implementation to optimize our compressed representation. Figure 11 shows our compression times for a single 4k material texture set with 9 channels, compared to a reference implementation in PyTorch [58]. Both implementations were evaluated on an NVIDIA RTX 4090 GPU for two different compression profiles.

Our custom implementation is approximately 10× faster than PyTorch, which is crucial for achieving practical compression times. We can generate a *preview* quality result in just under one minute for both configurations, with a difference of less than 1.5 dB compared to the maximum length of optimization (320k steps) in the 0.2 BPPC case. Moreover, for compression with the 1.0 BPPC profile, our implementation uses less than 2 GB of GPU memory, whereas PyTorch requires close to 18 GB, which is infeasible for many GPUs.

Traditional BCx compressors vary in speed, ranging from fractions of a second to tens of minutes to compress a single 4096 × 4096 texture [60], depending on quality settings. The median compression time for BC7 textures is a few seconds, while it is a fraction of a second for BC1 textures. This makes our method approximately an order of magnitude slower than a median BC7 compressor, but still faster than the slowest compression profiles.

6.5.2 Decompression. We evaluate real-time performance of our method by rendering a full-screen quad at 3840 × 2160 resolution textured with the *Paving Stone* set, which has 8 4k channels: diffuse albedo, normals, roughness, and ambient occlusion. The quad is lit by a directional light and shaded using a physically-based BRDF model [10] based on the Trowbridge–Reitz (GGX) microfacet distribution [76]. Results in Table 4 indicate that rendering with NTC via stochastic filtering (see Section 5.3) costs between 1.15 ms and 1.92 ms on a NVIDIA RTX 4090, while the cost decreases to 0.49 ms with traditional trilinear filtered BC7 textures. The performance is similar for all materials in our evaluation set, and independent of the output channel count, ranging from three to twelve. On the other hand, the varying number of features used across different compression profiles impacts the NTC performance. A higher number of features increases the sampling cost and the size of the network's first, input layer. We also implemented trilinear filtering for NTC by decompressing and filtering together eight texels and observed an 8× slowdown.

Although NTC is more expensive than traditional hardware-accelerated texture filtering, our results demonstrate that our method achieves high performance and is practical for use in real-time rendering. Furthermore, when rendering a complex scene in a fully-featured renderer, we expect the cost of our method to be partially hidden by the execution of concurrent work (e.g., ray tracing) thanks to the GPU latency hiding capabilities. The potential for latency hiding depends on various factors, such as hardware architecture, the presence of dedicated matrix-multiplication units that are otherwise under-utilized, cache sizes, and register usage. We leave investigating this for future work.

Table 5. Texture set quality as a function of material channel count.

channels	1	2	3	4	5	6	7	8	9
PSNR	36.22	29.97	29.34	28.96	29.57	28.68	28.22	28.43	28.71
BPPC	2.19	1.1	0.73	0.55	0.44	0.36	0.31	0.27	0.244

7 DISCUSSION

In this section, we discuss various aspects of our compression. First, we verify the original motivation for our work by analyzing compression across channels and mipmap levels. Following this, we present various limitations of our approach, including failure cases. Finally, we discuss future work and potential applications.

7.1 Compression Across Texture Channels and Mip Levels

Texture Channels. Table 5 shows results from our compression on a single texture set, with channel counts from 1 to 9. We selected the *Paving Stones* set as it is one of the most challenging texture set in our evaluation set. We keep the compressed representation size fixed, expecting to observe a significant reduction in PSNR in case of uncorrelated properties, due to entropy limitations imposed by information theory. Above two channels, we observe roughly constant quality, indicating the presence of significant cross-channel correlations, and suggesting that our model is able to effectively learn and exploit them. The quality is not monotonic with respect to the channel count, because we report averaged PSNR across all channels. Some channels are more challenging to compress than others, which leads to a PSNR increase when the introduced additional channel is easier to compress and more highly correlated with the previous ones. Overall, we achieve a similar low error across a large number of channels.

Mipmap Levels. Since our method shares a single decoder for all mipmap levels, we also analyzed the impact of compressing only mipmap level 0 with the 0.2 BPPC profile, and observed that the PSNR score was within 0.5 dB. This indicates a high level of feature reuse across mip levels.

7.2 Limitations

Failure Cases. Every lossy image or texture compression algorithm produces visual degradation at low bitrates. Typically, our method only results in mild blurring and color shifts. However, there are a few objectionable failure cases, which are presented in Figure 12. We observed that two of the failure cases (c and d) result from potential material authoring errors, namely misaligned texture channels and banding present in only a single channel, respectively, in the reference uncompressed textures. Our method relies heavily on channel correlation, and can be very sensitive to any alignment errors. We include a more detailed discussion of these failure cases in our supplementary material (Appendix D).

Uniform Resolution. Our method relies on storing all textures within a single compressed material at the same resolution. It is common practice for video game artists to store less visually important textures at smaller resolutions. Our method can assign different levels of importance to textures by weighting its contribution to the loss, but otherwise requires all the single material input textures to be resampled to the same resolution before compression.

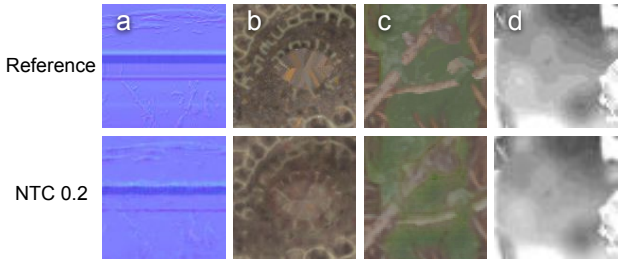

Fig. 12. Failure cases with our method. From left to right, a) removing fine details and noise, b) strong color shift, c) leaking of features between the texture channels, and d) removal of sharp staircase-like patterns.

Distance-Dependent Benefits. Our method operates at different compression rate profiles, but one of the more interesting configurations is the one with lowest bitrate (NTC 0.2). It demonstrates significantly more detail than BCx by enabling two higher resolution mipmap levels with the same storage. However, this increase in detail is not applicable at larger camera distances, when the additional mipmaps are no longer used.

Benefits Proportional to the Channel Count. Our method shows a high compression efficacy for materials with multiple channels. However, for lower channel counts, e.g., just RGB textures, our storage cost is similar at iso-quality (Table 5). This means that our method would lose some of its advantage if it was to be applied to single textures or regular images.

Decompression of All Channels. Our method always decompresses *all* material channels. Only the last output layer can be simplified for extraction of fewer textures, and it does not significantly reduce cost. This can be a limitation if different parts of texture set are used in different rendering contexts, for example, a partial depth prepass that only requires opacity maps or tessellation that only accesses a displacement map. In such cases, it might be a better choice to compress these textures separately using traditional methods.

Filtering Cost. Unrolled filtering is computationally expensive, and stochastic filtering can introduce flickering by increasing the burden on spatiotemporal reconstruction [87]. Literature shows that it is possible to create filterable neural representations directly [36], but we leave this for the future work.

Anisotropic Filtering. GPU texture samplers support *anisotropic* filtering, which improves the appearance of objects in the distance. However, a software implementation of anisotropic filtering with NTC would be prohibitively expensive for real-time rendering as it requires a large number of taps, each of which needs to be decoded.

7.3 Future Work

More Texture Types. Traditional GPU compression can support many types of textures, such as cube maps, 3D textures, and HDR textures. We have not investigated the feasibility of application to those, and leave it for future work.

Appearance-Based Training. Recently emerging inverse and neural rendering techniques allow to use an appearance-based loss function. Using the rendering error to drive the texture compression instead of the BRDF property similarity could allow for even more efficient content adaptation.

Inlined Material Evaluation. Our method targets compression of arbitrary, generic material textures that can be used by an analytical or a neural renderer. For the latter, there is no need to unpack materials to an intermediate representation. It is possible to fuse together decompression and BRDF evaluation into a single MLP. We leave this for future research.

Generative Textures and Materials. In our work, we target a faithful texture compression and preservation of existing detail. It is possible to expand it further and use *generative* approaches, where new plausible detail is generated upon zoom, despite not being present in the original texture. Using some form of generative super-resolution, it should be possible to generate multiple finer additional texture mipmaps, without ever storing them on disk or in memory.

Further Optimizations. Further improvements to the compression ratio and inference speed could be achieved through lower precision intermediate computations.

8 CONCLUSION

We have introduced a novel texture compression algorithm, targeting the increasing memory and fidelity requirements of modern computer graphics applications, and new, richer physically-based shading models that require many properties, commonly stored in textures. For high-performance texture accesses, it is of utmost importance to be able to spatially access the textures anywhere at a small cost, which is often referred to as the random access property. We have shown that very high compression rates can be achieved even without sacrificing local and random access.

By compressing many channels and mipmap levels together, the quality of our algorithm's low bitrate results surpasses that of state-of-the-art industry standards, such as JPEG XL and AVIF that are substantially more complex methods, without requiring entropy coding.

By utilizing matrix multiplication intrinsics available in the off-the-shelf GPUs, we have shown that decompression of our textures introduces only a modest timing overhead as compared to simple BCx algorithms (which executes in custom hardware), possibly making our method practical in disk- and memory-constrained graphics applications.

We hope our work will inspire the creation of highly compressed neural representations for use in other areas of real-time rendering, as a means of achieving cinematic quality.

ACKNOWLEDGMENTS

We are grateful for the help of our colleagues on this project: John Burgess, for many valuable suggestions and ideas; Yong He, Patrick Neill, Justin Holewinski, and Petrik Clarberg for their help with Slang programming language, driver support for fast matrix multiplication, and performance evaluation; Toni Bratincevic and Andrea Weidlich for the Inkwell asset and its material adapted to the Falcor framework. Finally, we would like to thank Lennart Demes, author of the ambientCG website, for providing a public domain material database that we used to evaluate the efficacy of our algorithm. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

[1] Jyrki Alakuijala, Robert Obryk, Ostap Stoliarchuk, Zoltan Szabadka, Lode Vandevenne, and Jan Wassenberg. 2017. Guetzli: Perceptually Guided JPEG Encoder. *arXiv:1703.04421* (2017).

[2] Jyrki Alakuijala, Ruud Van Asseldonk, Sami Boukortt, Martin Bruse, Iulia-Maria Comșa, Moritz Firsching, Thomas Fischbacher, Evgenii Kliuchnikov, Sebastian Gomez, Robert Obryk, et al. 2019. JPEG XL Next-Generation Image Compression Architecture and Coding Tools. In *Applications of Digital Image Processing XLII*, Vol. 11137. SPIE, 112–124.

[3] Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D Fairchild. 2020. ꟻLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 2 (2020), 1–23.

[4] Thomas Arcila. 2022. FidelityFX Super Resolution 2.0. In *Game Developers Conference*.

[5] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. 2017. End-to-end Optimized Image Compression. In *International Conference on Learning Representations*.

[6] Johannes Ballé, Philip Chou, David Minnen, Saurabh Singh, Nick Johnston, Eirikur Agustsson, Sung Hwang, and George Toderici. 2021. Nonlinear Transform Coding. *IEEE Journal of Selected Topics in Signal Processing* 15, 2 (2021), 339–353.

[7] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. 2018. Variational Image Compression with a Scale Hyperprior. In *International Conference on Learning Representations*.

[8] Yochai Blau and Tomer Michaeli. 2018. The Perception-Distortion Tradeoff. In *IEEE Conference on Computer Vision and Pattern Recognition*. 6228–6237.

[9] Benjamin Bross, Ye-Kui Wang, Yan Ye, Shan Liu, Jianle Chen, Gary J. Sullivan, and Jens-Rainer Ohm. 2021. Overview of the Versatile Video Coding (VVC) Standard and its Applications. *IEEE Transactions on Circuits and Systems for Video Technology* 31, 10 (2021), 3736–3764. https://doi.org/10.1109/TCSVT.2021.3101953

[10] Brent Burley and Walt Disney Animation Studios. 2012. Physically-Based Shading at Disney. In *ACM SIGGRAPH course*.

[11] Graham Campbell, Thomas A. DeFanti, Jeff Frederiksen, Stephen A. Joyce, Lawrence A. Leske, John A. Lindberg, and Daniel J. Sandin. 1986. Two Bit/Pixel Full Color Encoding. *Computer Graphics (SIGGRAPH)* 20, 4 (1986), 215–223.

[12] Thomas Chambon, Eric Heitz, and Laurent Belcour. 2021. Passing Multi-Channel Material Textures to a 3-Channel Loss. In *ACM SIGGRAPH Talks*. Article 12, 2 pages.

[13] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, Ching-Han Chiang, Yunqing Wang, Paul Wilkins, Jim Bankoski, Luc Trudeau, Nathan Egge, Jean-Marc Valin, Thomas Davies, Steinar Midtskogen, Andrey Norkin, and Peter de Rivaz. 2018. An Overview of Core Coding Tools in the AV1 Video Codec. In *Picture Coding Symposium*. IEEE, 41–45.

[14] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. 2020. Learned Image Compression With Discretized Gaussian Mixture Likelihoods and Attention Modules. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[15] Shin-Fang Chng, Sameera Ramasinghe, Jamie Sherrah, and Simon Lucey. 2022. Gaussian activated neural radiance fields for high fidelity reconstruction and pose estimation. In *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXXIII*. Springer, 264–280.

[16] Adrian Courrèges. 2020. Graphics Studies Compilation. http://www.adriancourreges.com/blog/2020/12/29/graphics-studies-compilation/

[17] I.A. Cunningham, M.S. Westmore, and A. Fenster. 1995. A Stochastic Convolution that Describes both Image Blur and Image Noise using Linear Systems Theory. In *International Conference of the Engineering in Medicine and Biology Society*, Vol. 1. 555–556.

[18] E. Delp and O. Mitchell. 1979. Image Compression Using Block Truncation Coding. *IEEE Transactions on Communications* 27, 9 (1979), 1335–1342.

[19] Manfred Ernst, Marc Stamminger, and Gunther Greiner. 2006. Filter Importance Sampling. In *2006 IEEE Symposium on Interactive Ray Tracing*. 125–132.

[20] Simon Fenney. 2003. Texture Compression Using Low-Frequency Signal Modulation. In *Graphics Hardware*. 84–91.

[21] The Khronos Group. 2019. KHR_shader_subgroup. https://github.com/KhronosGroup/GLSL/blob/master/extensions/nv/GLSL_NV_cooperative_matrix.txt

[22] The Khronos Group. 2019. NV_cooperative_matrix. https://github.com/KhronosGroup/GLSL/blob/master/extensions/nv/GLSL_NV_cooperative_matrix.txt

[23] The Khronos Group. 2023. Vulkan 1.3.246 - A Specification. https://registry.khronos.org/vulkan/specs/1.3/html/vkspec.html

[24] J. Hasselgren, J. Munkberg, M. Salvi, A. Patney, and A. Lefohn. 2020. Neural Temporal Adaptive Sampling and Denoising. *Computer Graphics Forum* 39, 2 (2020), 147–155.

[25] Yong He, Kayvon Fatahalian, and T. Foley. 2018. Slang: Language Mechanisms for Extensible Real-Time Shading Systems. *ACM Transactions on Graphics* 37, 4, Article 141 (2018).

[26] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:1606.08415* (2016).

[27] Nikolai Hofmann, Jon Hasselgren, Petrik Clarberg, and Jacob Munkberg. 2021. Interactive Path Tracing and Reconstruction of Sparse Volumes. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 4, 1, Article 5 (apr 2021).

[28] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer Deedforward Networks are Universal Approximators. *Neural Networks* 2, 5 (1989), 359–366.

[29] A. Howard, M. Sandler, B. Chen, W. Wang, L. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le. 2019. Searching for MobileNetV3. In *International Conference on Computer Vision*. 1314–1324.

[30] Stephanie Hurlburt and Rich Geldreich. 2022. Basis Universal Supercompressed GPU Texture Codec. https://github.com/BinomialLLC/basis_universal

[31] Intel. 2022. Intel® Arc™ Graphics Developer Guide for Real-Time Ray Tracing in Games. https://www.intel.com/content/www/us/en/developer/articles/guide/real-time-ray-tracing-in-games.html

[32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *IEEE Conference on Computer Vision and Pattern Recognition*. 2704–2713.

[33] Robert Kawiak, Hisham Chowdhury, Rense de Boer, Gabriel N. Ferreira, and Lucas Xavier. 2022. Intel Xe Super Sampling (XeSS)—an AI-based Upscaling for Real-Time Rendering. In *Game Developers Conference*.

[34] Diederik P Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980* (2014).

[35] G. Knittel, A. Schilling, A. Kugler, and W. Straßer. 1996. Hardware for Superior Texture Performance. *Computers & Graphics* 20, 4 (1996), 475–481.

[36] Alexandr Kuznetsov, Krishna Mullia, Zexiang Xu, Miloš Hašan, and Ravi Ramamoorthi. 2021. NeuMIP: Multi-Resolution Neural Materials. *ACM Transactions on Graphics* 40, 4 (2021).

[37] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. 2017. Photo-Realistic Single Image Super-Resolution using a Generative Adversarial Network. In *IEEE Conference on Computer Vision and Pattern Recognition*. 4681–4690.

[38] David B Lindell, Dave Van Veen, Jeong Joon Park, and Gordon Wetzstein. 2022. Bacon: Band-limited coordinate networks for multiscale scene representation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 16252–16262.

[39] Edward Liu. 2022. DLSS 2.0 - Image Reconstruction for Real-Time Rendering with Deep learning. In *Game Developers Conference*.

[40] Haojie Liu, Tong Chen, Peiyao Guo, Qiu Shen, Xun Cao, Yao Wang, and Zhan Ma. 2019. Non-local Attention Optimized Deep Image Compression. *ArXiv* abs/1904.09757 (2019).

[41] Stephen Lombardi, Tomas Simon, Jason Saragih, Gabriel Schwartz, Andreas Lehrmann, and Yaser Sheikh. 2019. Neural Volumes: Learning Dynamic Renderable Volumes from Images. *arXiv:1906.07751* (2019).

[42] Ilya Loshchilov and Frank Hutter. 2017. SGDR: Stochastic Gradient Descent with Warm Restarts. In *International Conference on Learning Representations*.

[43] Fabian Mentzer, George D Toderici, Michael Tschannen, and Eirikur Agustsson. 2020. High-Fidelity Generative Image Compression. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. 11913–11924.

[44] Microsoft. 2015. Direct3D 11.3 Functional Specification. https://microsoft.github.io/DirectX-Specs/d3d/archive/D3D11_3_FunctionalSpec.htm

[45] Microsoft. 2020. Texture Block Compression in Direct3D 11. https://learn.microsoft.com/en-us/windows/win32/direct3d11/texture-block-compression-in-direct3d-11

[46] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2021. Nerf: Representing Scenes as Neural Radiance Fields for View Synthesis. *Commun. ACM* 65, 1 (2021), 99–106.

[47] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Transactions on Graphics* 41, 4 (2022), 1–15.

[48] Thomas Müller, Fabrice Rousselle, Jan Novák, and Alexander Keller. 2021. Real-time Neural Radiance Caching for Path Tracing. *ACM Transactions on Graphics* 40, 4 (2021), 1–16.

[49] Jacob Munkberg, Petrik Clarberg, Jon Hasselgren, and Tomas Akenine-Möller. 2006. High Dynamic Range Texture Compression for Graphics Hardware. *ACM Transactions on Graphics* 25, 3 (2006), 698–706.

[50] Krzysztof Narkowicz. 2016. ACES Filmic Tone Mapping Curve. https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/, January 6.

[51] David Neubelt and Matt Pettineo. 2013. Crafting a Next-Gen Material Pipeline for the Order: 1886. *Physically Based Shading in Theory and Practice, SIGGRAPH Courses* (2013).

[52] NVIDIA. 2022. NVVM IR Specification. https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html

[53] NVIDIA. 2022. Shader Execution Reordering. https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering/

[54] NVIDIA. 2023. Matrix Fragments for mma.m16n8k16. https://docs.nvidia.com/cuda/parallel-thread-execution/#warp-level-matrix-fragment-mma-16816-float

[55] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. 2012. Adaptive Scalable Texture Compression. In *High-Performance Graphics*. 105–114.

[56] Yaobin Ouyang, Shiqiu Liu, Markus Kettunen, Matt Pharr, and Jacopo Pantaleoni. 2021. ReSTIR GI: Path Resampling for Real-Time Path Tracing. *Computer Graphics Forum* 40, 8 (2021), 17–29.

[57] Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. 2019. DeepSDF: Learning Continuous Signed Distance Functions for Shape Representation. In *Conference on Computer Vision and Pattern Recognition*.

[58] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32.

[59] Gregg William Perkins and Santiago Echeverry. 2022. Virtual Production in Action: A Creative Implementation of Expanded Cinematography and Narratives. In *ACM SIGGRAPH 2022 Posters*. Article 21.

[60] Aras Pranckevičius. 2020. Texture Compression in 2020. https://aras-p.info/blog/2020/12/08/Texture-Compression-in-2020/

[61] Oren Rippel and Lubomir Bourdev. 2017. Real-Time Adaptive Image Compression. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 2922–2930.

[62] Kimmo Roimela, Tomi Aarnio, and Joonas Itäranta. 2006. High Dynamic Range Texture Compression. *ACM Transactions on Graphics* 25, 3 (2006), 707–712.

[63] Vishwanath Saragadam, Daniel LeJeune, Jasper Tan, Guha Balakrishnan, Ashok Veeraraghavan, and Richard G Baraniuk. 2023. WIRE: Wavelet Implicit Neural Representations. *arXiv preprint arXiv:2301.05187* (2023).

[64] Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. 2020. Implicit neural representations with periodic activation functions. *Advances in Neural Information Processing Systems* 33 (2020), 7462–7473.

[65] A. Skodras, C. Christopoulos, and T. Ebrahimi. 2001. The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine* 18, 5 (2001), 36–58.

[66] Jacob Ström and Tomas Akenine-Möller. 2005. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Graphics Hardware*. 63–70.

[67] Jacob Ström and Martin Pettersson. 2007. ETC2: Texture Compression Using Invalid Combinations. In *Graphics Hardware*. 49–54.

[68] Towaki Takikawa, Alex Evans, Jonathan Tremblay, Thomas Müller, Morgan McGuire, Alec Jacobson, and Sanja Fidler. 2022. Variable Bitrate Neural Fields. In *ACM SIGGRAPH 2022 Conference Proceedings*. 1–9.

[69] Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. 2020. Fourier Features let Networks Learn High Frequency Functions in Low Dimensional Domains. *Advances in Neural Information Processing Systems* 33 (2020), 7537–7547.

[70] Natalya Tatarchuk, Jonathan Dupuy, Thomas Deliot, Daniel Wright, Krzysztof Narkowicz, Patrick Kelly, Aleksander Netzel, and Tiago Costa. 2022. Advances in Real-Time Rendering in Games: Part I. In *ACM SIGGRAPH Courses*. Article 18.

[71] Natalya Tatarchuk, Timothy Lottes, Kleber Garcia, Thomas Deliot, Jonathan Dupuy, Kees Rijnen, Xiaoling Yao, Brian Karis, Graham Wihlidal, Rune Stubbe, and Ari Silvennoinen. 2021. Advances in Real-Time Rendering in Games: Part I. In *ACM SIGGRAPH Courses*.

[72] Ayush Tewari, Justus Thies, Ben Mildenhall, Pratul Srinivasan, Edgar Tretschk, Wang Yifan, Christoph Lassner, Vincent Sitzmann, Ricardo Martin-Brualla, Stephen Lombardi, et al. 2022. Advances in Neural Rendering. In *Computer Graphics Forum*, Vol. 41. 703–735.

[73] Lucas Theis, Wenzhe Shi, Andrew Cunningham, and Ferenc Huszár. 2017. Lossy Image Compression with Compressive Autoencoders. In *International Conference on Learning Representations*.

[74] Justus Thies, Michael Zollhöfer, and Matthias Nießner. 2019. Deferred Neural Rendering: Image Synthesis using Neural Textures. *ACM Transactions on Graphics* 38, 4 (2019), 1–12.

[75] Manu Mathew Thomas, Gabor Liktor, Christoph Peters, Sungye Kim, Karthik Vaidyanathan, and Angus G. Forbes. 2022. Temporally Stable Real-Time Joint Neural Denoising and Supersampling. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 5, 3, Article 21 (jul 2022).
[76] T. S. Trowbridge and K. P. Reitz. 1975. Average Irregularity Representation of a Rough Surface for Ray Reflection. *Journal of the Optical Society of America* 65, 5 (May 1975), 531–536.
[77] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. 2018. Deep Image Prior. In *IEEE Conference on Computer Vision and Pattern Recognition*. 9446–9454.
[78] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. 2017. Neural Discrete Representation Learning. In *International Conference on Neural Information Processing Systems*.
[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* 30 (2017).
[80] Gregory K Wallace. 1991. The JPEG Still Picture Compression Standard. *Commun. ACM* 34, 4 (1991), 30–44.
[81] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. 2004. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Transactions on Image Processing* 13, 4 (2004), 600–612.
[82] Wikipedia. 2022. S3 Texture Compression. https://en.wikipedia.org/wiki/S3_Texture_Compression (accessed on 2022-11-14).
[83] Alan Wolfe, Nathan Morrical, Tomas Akenine-Möller, and Ravi Ramamoorthi. 2022. Spatiotemporal Blue Noise Masks. In *Eurographics Symposium on Rendering*.
[84] Bartlomiej Wronski. 2020. Dimensionality Reduction for Image and Texture Set Compression. https://bartwronski.com/2020/05/21/dimensionality-reduction-for-image-and-texture-set-compression/
[85] Bartlomiej Wronski. 2021. Neural Material (De)compression – Data-Driven Nonlinear Dimensionality Reduction. https://bartwronski.com/2021/05/30/neural-material-decompression-data-driven-nonlinear-dimensionality-reduction/
[86] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *IEEE International Parallel and Distributed Processing Symposium*. 634–643.
[87] Lei Yang, Shiqiu Liu, and Marco Salvi. 2020. A Survey of Temporal Antialiasing Techniques. *Computer Graphics Forum* 39, 2 (2020), 607–621.
[88] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang. 2018. The Unreasonable Effectiveness of Deep Features as a Perceptual Metric. In *IEEE Conference on Computer Vision and Pattern Recognition*. 586–595.
[89] Maria Zontak and Michal Irani. 2011. Internal Statistics of a Single Natural Image. In *Conference on Computer Vision and Pattern Recognition*. 977–984.

A HANDLING DIVERGENCE

Using matrix acceleration for the neural network requires all SIMD lanes to be active and network weights to be uniform across the SIMD lanes. However, in some scenarios like ray tracing, rays from the same SIMD group may hit different materials or miss geometry altogether. When querying ray-scene intersections from a compute shader, users can control the execution mask, ensuring all SIMD lanes are active during network evaluation. Conversely, in hit or miss shaders, users lack control over the shader execution mask. In these cases, the users can query the execution mask and enable tensor acceleration when all lanes are active, otherwise a fallback path without tensor acceleration is necessary.

The following example code shows how divergence can be handled inside a hit shader by enabling matrix acceleration when all lanes are active, and by iterating over unique sets of network parameter offsets, which are broadcast across all SIMD lanes to make them uniform. SIMD occupancy in a complex scene with a large number of materials can potentially be improved with techniques like SER [53] and TSU [31]. We leave this evaluation for future work.

```
Outputs runNetwork(Inputs x, uint paramOffsets) {
  // Check if all lanes are active.
  if (WaveActiveCountBits(true) == WaveGetLaneCount()) {
    uint mask = -1;
    uint lane = 0;
    // Iterate over unique network parameters in the SIMD group.
    for (; mask ;) {
      // Broadcast the parameter offset across SIMD lanes.
      uint offset = WaveReadLaneAt(paramOffsets, lane);
      bool matchingLanes = offset == paramOffsets;

      // Evaluate the MLP with matrix acceleration.
      Outputs y = MLP(x, offset);

      // Store the outputs for matching lanes.
      storeOutputs(y, matchingLanes);

      // Clear the evaluated lanes.
      mask -= WaveActiveBallot(matchingLanes).x;
      lane = firstbitlow(mask);
    }
  } else {
    // Fallback without matrix acceleration.
  }
}
```

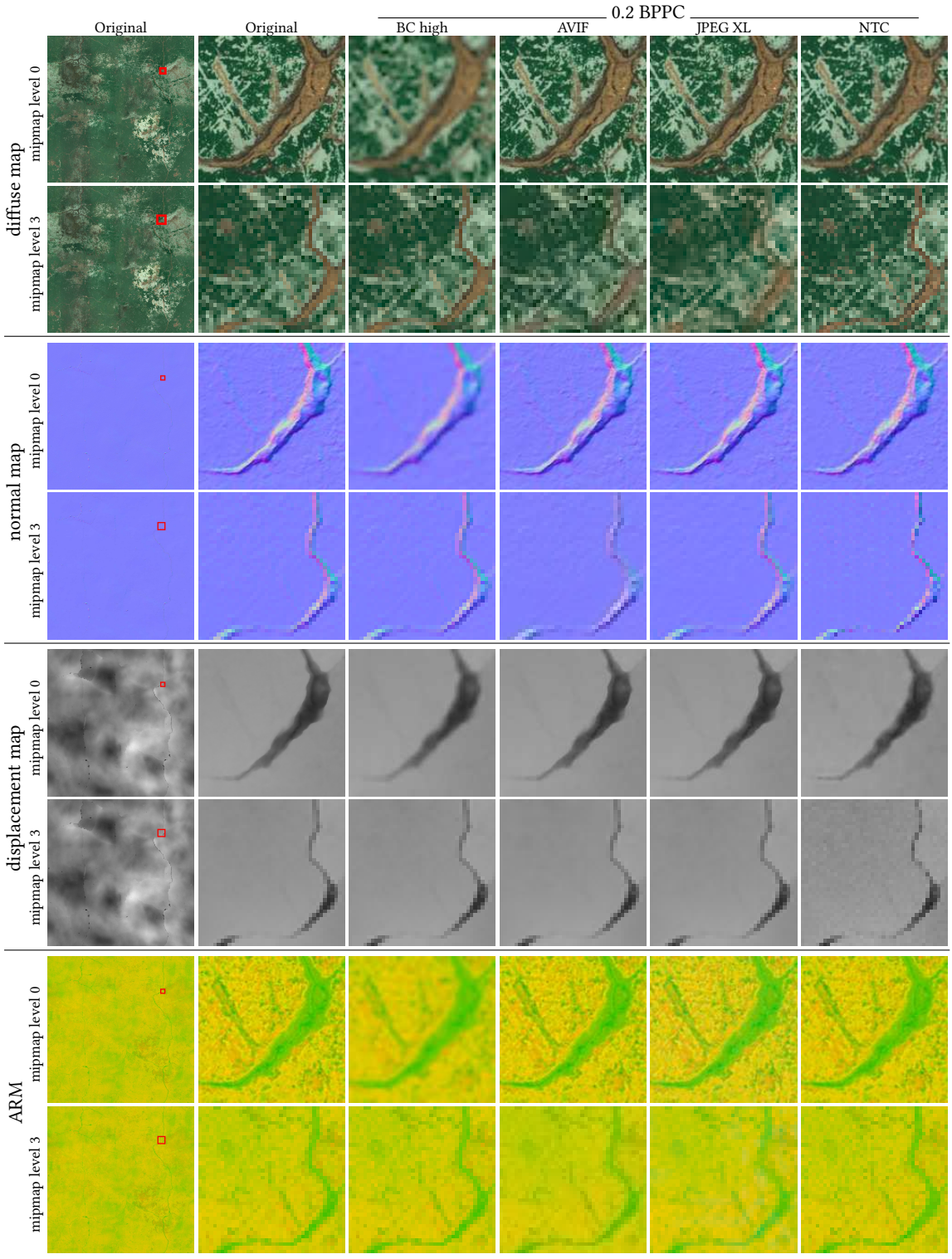

Fig. 13. Comparison of different methods at 0.2 BPPC, where we selected to show the texture set for which NTC's PSNR was closest to its average PSNR over all texture sets in our evaluation dataset. Recall that neither AVIF nor JPEG XL provide random access to the texture data. Furthermore, to achieve an approximately iso-storage comparison, mipmap level 0 images for BC high were created by bilinear upsampling of its mipmap level 2. The textures were retrived from https://polyhaven.com/. Additional examples are available in our supplementary material.

SUPPLEMENTARY TO RANDOM-ACCESS NEURAL COMPRESSION OF MATERIAL TEXTURES

B RENDERED QUALITY

In addition to the first figure in the main paper, Figures 14 and 15 show examples of rendered images using high-quality non-compressed textures, textures compressed with our method (using the lowest BPPC profile, NTC 0.2), and textures compressed with BC high. For the latter, the two highest-resolution mip levels were created through bilinear upscaling of the third mip level in order to obtain an iso-storage comparison to our method. The rendered images, supported by the error images and values, show that NTC achieves higher quality rendered images than the iso-storage version of BC high. This is especially noticeable in Figure 15, which also gives an indication that NTC does well on textures with text.

C ADDITIONAL QUANTITATIVE RESULTS

The following two subsections demonstrate how the quality of NTC's compressed images changes over the mipmap chain (Section C.1) and how it differs for various texture types (Section C.2).

C.1 Mipmap Quality

The performance of a compression technique can vary based on the frequency spectrum of the image and therefore perform differently across mip levels. In Figure 21, we compare the per-mip-level PSNR scores of NTC 0.2 to the BC high algorithm. Since there is a 16× difference in storage cost between the algorithms, an iso-storage comparison was conducted, resulting in the omission of the first two mip levels for BC compression. PSNR values with our method are either comparable to, or higher than BC depending on the mip level, except for mip levels two and three where NTC shows slightly worse PSNR scores.

C.2 Different Texture Type Compression Quality

Figure 20 presents the PSNR scores of NTC 0.20, computed on different types of textures in the material texture set, such as diffuse, normals, etc. Given the similarities in data content between certain texture types, such as ARM and ORM textures, as well as gloss and specular textures, the results for these pairs were concatenated. Additionally, texture types that occurred only once in our data set (see Section J) were excluded. The results indicate that our proposed method is able to compress different texture types at similar levels of quality.

D COMPRESSION ARTIFACTS

Every texture compression algorithm degrades quality differently. Particularly visible quality differences are called *compression artifacts* and we present a few typical examples in Figure 16. Block-based compression methods commonly exhibit visible block artifacts (inset a). Methods that rely on heavy quantization tend to exhibit banding artifacts, as illustrated in insets (b) and (c), and are often characterized by a visible discoloration towards green or purple hues, resulting from higher chroma quantization (inset b). These artifacts are highly perceptible to the human eye, and modern image compression techniques such as AVIF and JPEG XL, have prioritized their removal, by producing blurry images instead (inset d).

Since our feature vectors are quantized down to *two* or *four* bits per feature, we specifically check for the presence of banding artifacts by compressing a synthetic gradient texture and do not observe noticeable banding artifacts as shown in Figure 19. We attribute this to the combination of smooth, bilinear interpolation and the higher frequency learned interpolation (see Sections 4.1 and 4.3).

Figure 13 demonstrates the absence of visually objectionable artifacts with our compression method on an average case. In contrast, we observe the presence of block artifacts on the ARM texture with BC, especially for mip level 0. AVIF and JPEG XL produce sharper results than NTC for the diffuse texture at mip level 0 but are significantly blurrier or show some discoloration (JPEG XL) at mip level 3. This is likely because with these methods, different mip levels are compressed separately, and their spectral content does not necessarily reduce proportionally to the resolution. These observations make a strong case for jointly compressing mip levels as we do with NTC. Potentially, we could compress AVIF and JPEG XL by allocating different rates for each mip as well as each texture, while maintaining the overall storage constant. However, determining suitable rates can be challenging, particularly as it can be material specific. We do not include comparisons with heterogeneous rates for this reason, and also because our proposed method does not aim to compete directly with these image compression techniques.

On average, our method produces results that are a bit blurrier and sometimes less saturated than the uncompressed reference, but significantly better than BCx compression. There are, however, some more objectionable failure cases presented in Figure 12 in the main paper. In example a) we observe strong distortion of the normal map of the *Ticket Machine* texture. The compressed texture is very flat, mostly sparse, and our optimization procedure fails to reconstruct subtle details properly. In example b) (albedo of the *dragon atlas* material), we observe discoloration of the texture. Since our approach is specialized for each material, it can adapt to high frequency content, such as detailed normal maps, or large color variations. However, the most challenging materials have both kinds of features. In such cases, we cannot reconstruct both features equally well given the low BPPC rates. Typically, we observe that details and normal map features are favored by our optimization because of their higher variance, at the cost of other material textures. It is possible to balance the quality of material textures by adjusting the loss function (Section 4.5). In scenarios where the material textures have a fixed set of semantics, we can apply more robust texture specific optimization, such as a loss in chrominance space, or optimization based on appearance using differentiable renderering. The last two failure cases (c and d) in Figure 12 correspond to unusual data in the source materials. In example c) (*Pine Forest Ground* texture), the normal maps seem to be misaligned with the albedo maps, producing leakage of details between channels. Example d) (metalness map of the *Metal Plates* texture) shows strong banding in the source (reference) texture, present only in a single material channel, and our method blurs it and correlates with the other material channels.

Figures 17 and 18 show results of texture compression at other BPPC targets than the ones shown in the main paper. We note that the texture used to generate Figure 17 (*Pine Forest Ground*) is half the size (4096×4096) of the texture used to generate Figure 18 (*denim*) and the corresponding figure in the main paper. In Figure 17, we

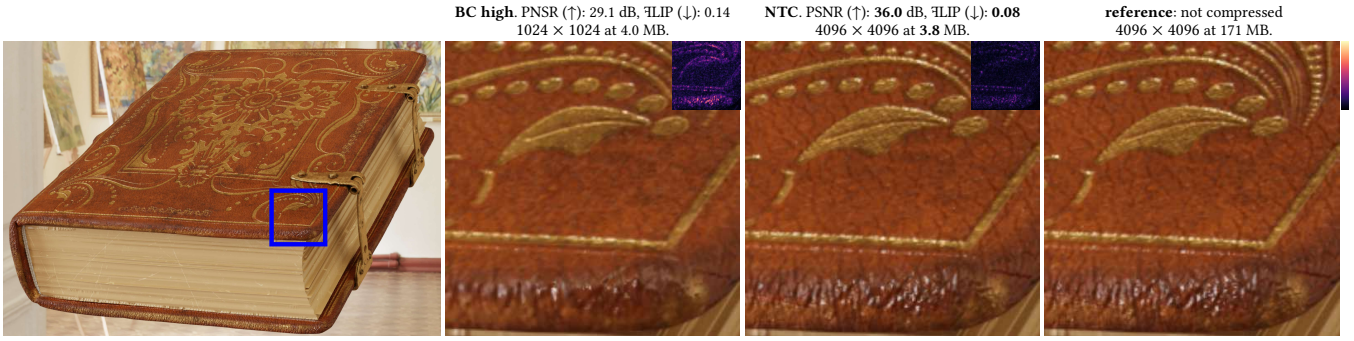

Fig. 14. A rendered image of a closed book. The cutouts demonstrate quality using, from left to right, GPU-based texture formats (BC high) at 1024 × 1024 resolution, our neural texture compression (NTC) , and high-quality reference textures. Note that NTC provides two additional mipmap levels over BC high, despite it using slightly less memory. The metrics, PSNR and ꟻLIP, were computed for the cutouts and are shown above the respective image. The ꟻLIP error images, whose brightness is proportional to error, are shown in the upper right corners.

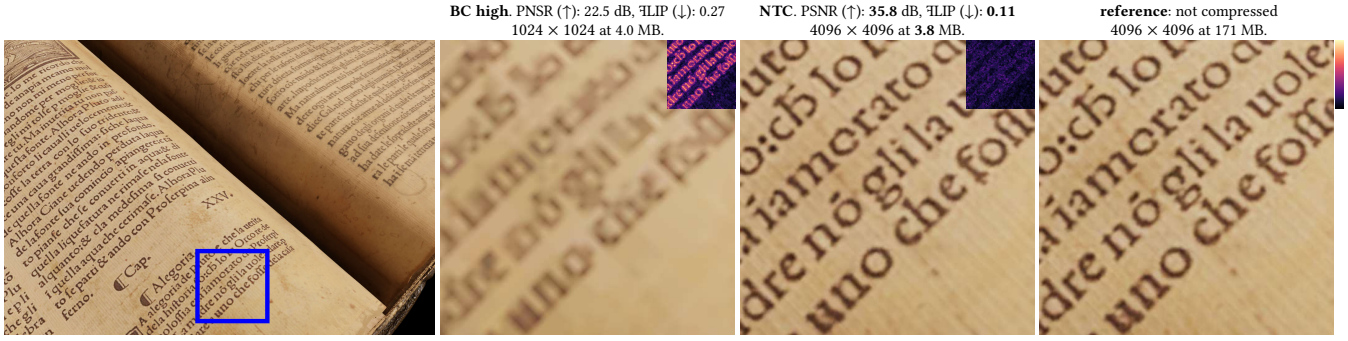

Fig. 15. A rendered image of an open book. See Figure 14's caption for details.

compare to the medium-low-rate JPEG XL and AVIF configurations, as well as the ASTC compressor (using tiles of size 12 × 12, as those lead to a mean BPPC closer to 0.50 for the evaluation data set compared to tiles of size 10 × 10). Figure 18 compares the medium-bitrate compressors. In the medium-bitrate case, it is difficult to spot any differences between the compressed textures and the reference. For the medium-low-rate case, differences become visible, most notably for the higher level mipmap where ASTC 12 × 12 and AVIF show block artifacts. Here, NTC 0.5 show slight color changes in the diffuse texture. An added challenge of this texture set is that the normal map is not aligned with the remaining textures.

E COMPARISON TO VECTOR QUANTIZATION

Vector quantization (VQ) is an alternate approach [78] to discretizing the features, where each cell in a feature grid maps to an entry in a learned codebook or dictionary. During inference, the feature vectors can be replaced by a codebook index stored per grid cell. Unfortunately, the codebook size grows exponentially with the bitrate, making it prohibitively expensive to learn a codebook for higher quality levels. Therefore, in order to compare VQ with scalar quantization (SQ), we limit the size of the dictionary to 256 entries and assume multiple dictionaries, such that the overall storage size is the same. We only apply vector quantization to the higher resolution grid G_0, which is quantized to a smaller number of bits, while G_1 always uses scalar quantization with 12 channels and 4 bits.

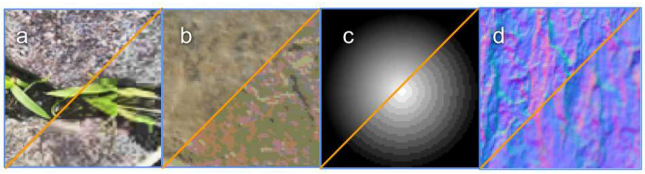

Fig. 16. Examples of typical compression artifacts. **a)** visible blocks **b)** posterization and discoloration **c)** gradient banding **d)** detail loss and bluriness.

Table 7 shows a comparison of SQ and VQ for a 4k texture using our lowest bitrate configuration (NTC 0.2) where each grid cell in G_0 stores 16 bits. In the case of SQ, we use 8 channels which are quantized to 2 bits while in the case of VQ, we use two 256-entries dictionaries, which are referenced by two 8-bit indices respectively. We compare SQ against two variants of VQ: VQ-8 has 8 channels per dictionary entry, which is similar to the size of the feature vector used in SQ, while VQ-16 uses 16 channels per dictionary entry and a correspondingly larger input layer in the decoder network. The PSNR for all three quantization options are within 0.5 dB of each other. VQ-8 has slightly lower PSNR than SQ, while VQ-16 has a slightly higher PSNR, but with a higher cost for the input layer of the network. The training time for both VQ-8 and VQ-16 is more than 2.5× that of SQ. Given its simplicity and the significantly shorter training time, we choose scalar quantization for compression.

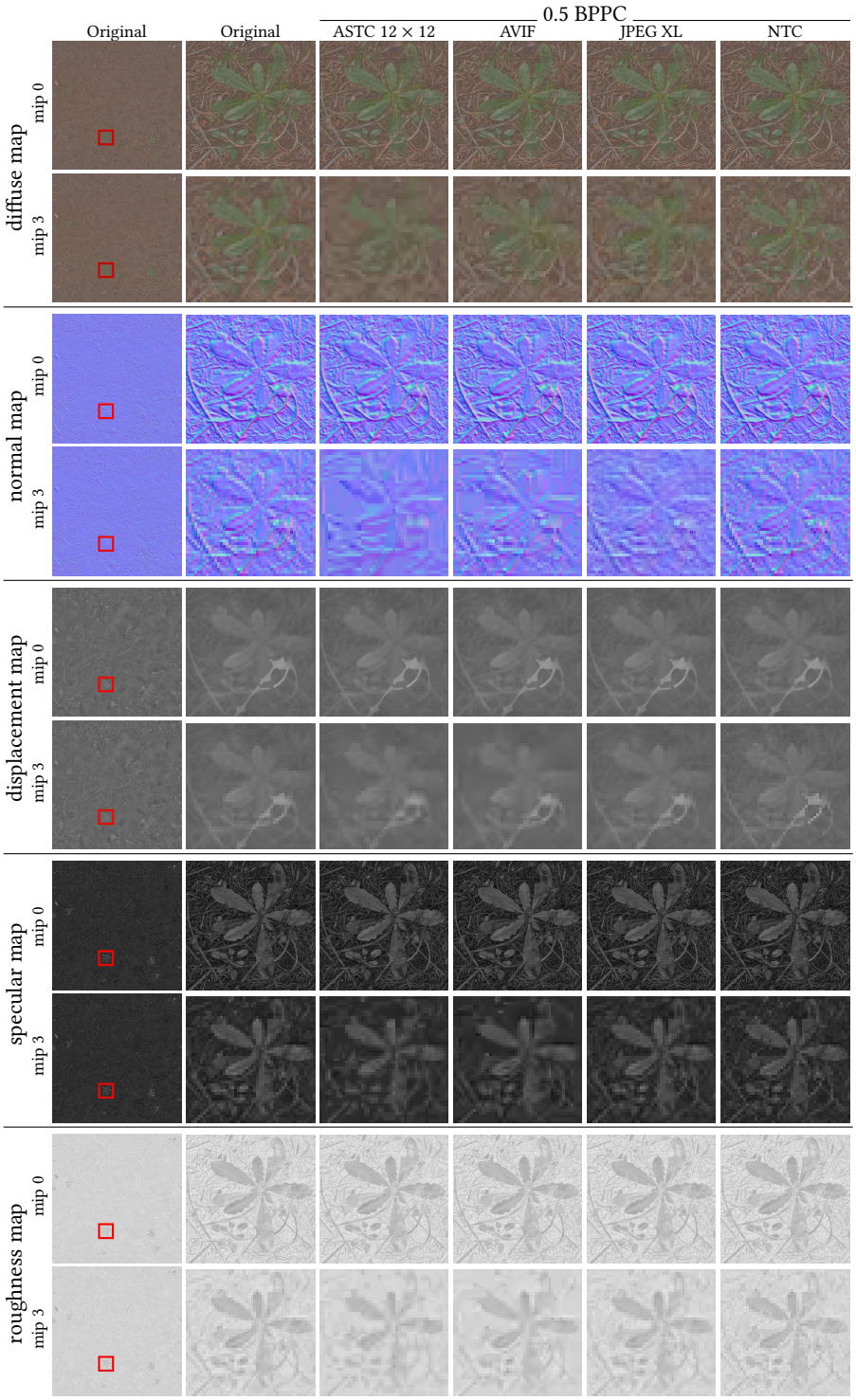

Fig. 17. Comparison of different methods at 0.5 BPPC, where we selected to show a texture set for which NTC's PSNR was close to its average PSNR over all texture sets in our 20 texture evaluation dataset. Recall that neither AVIF nor JPEG XL provide random access to the texture data. For visualization purposes, the diffuse images were exposure compensated with factor −1.0 and tone mapped with ACES [50]. Textures retrived from https://kaimoisch.com/free-textures/.

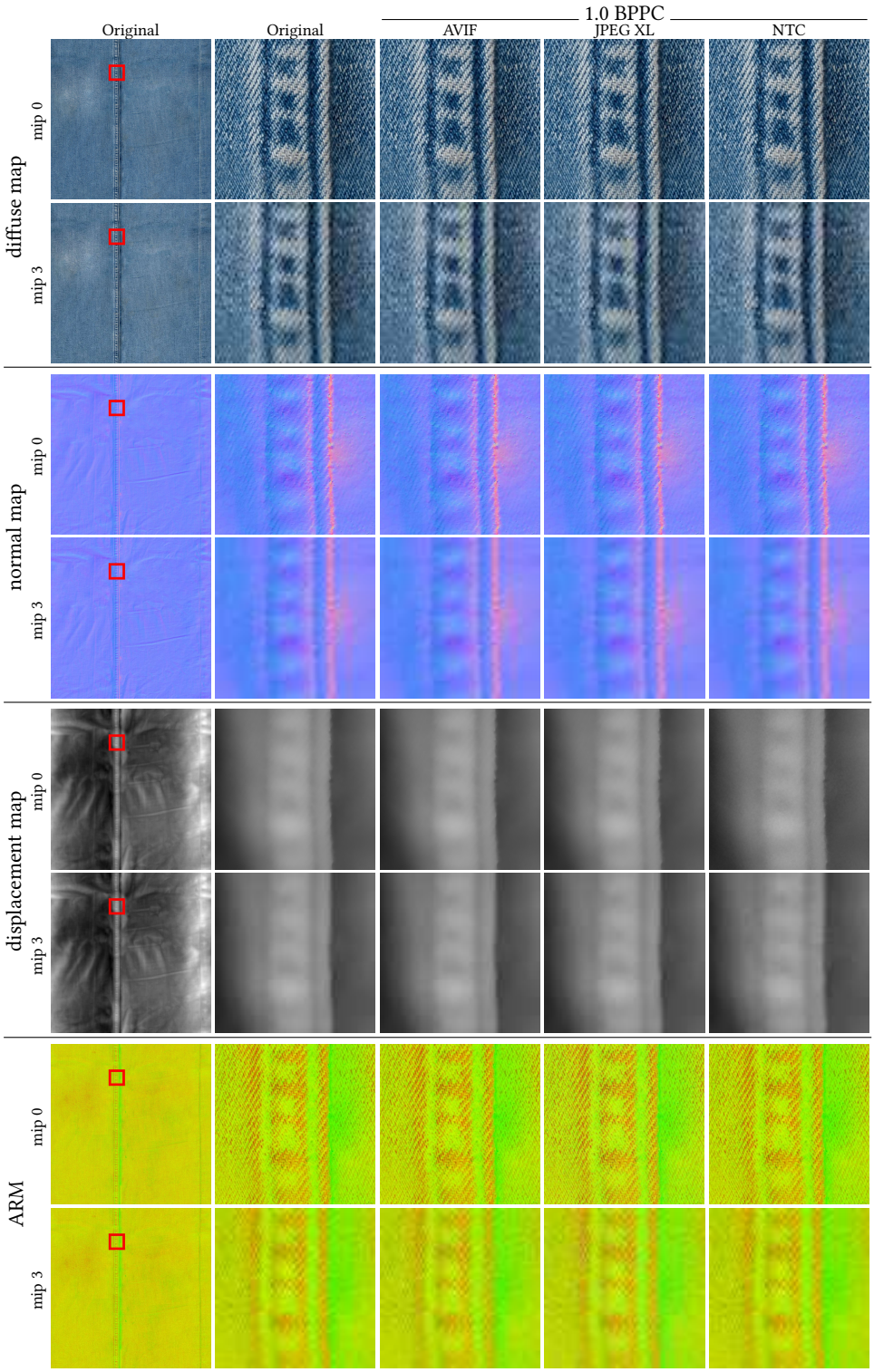

Fig. 18. Comparison of different methods at 1.0 BPPC, where we selected to show a texture set for which NTC's PSNR was close to its average PSNR over all texture sets in our 20 texture evaluation dataset. Recall that neither AVIF nor JPEG XL provide random access to the texture data. Textures retrived from https://polyhaven.com/.

F STORAGE COST

For completeness, we list the storage cost of NTC in Table 6 for our different compression profiles and for different texture set resolutions.

G FAILED EXPERIMENTS

In the course of developing our method, we evaluated a few alternative methods for neural compression. We found that these methods, which are characterized by either increased complexity or inferior quality, were unsuitable for the task of texture compression of material textures. We present these findings below.

Warped Grids. Prior work [41] proposes to warp volumes with a non-uniform transformation for better resource utilization compared to a uniform grid. We hypothesized that a similar approach, applied to images, could achieve some of the benefits of nonuniform bit allocation of entropy coding. We found that the inclusion of warping grids led to an increase in PSNR scores between 0.1 and 0.9 dB, depending on the scale of the warping grid used. However, after compressing and quantizing the warp grid, all the observed benefits could be achieved by simply allocating similar additional amount of storage to our latent grids.

Nonuniform Quantization. We empirically observed that, prior to quantization, the distribution of our grid values closely resembles a truncated normal distribution. We tried adopting a normally distributed quantization scheme, but did not observe any quality improvement. We attribute this outcome to the fine-tuning of network weights after the freezing of the latent grids, which might compensate for the sub-optimal quantization distribution (see Section 4.2).

H USAGE OF OTHER COMPRESSORS

In this section, we describe which compressors we compare to and their parameters. Note that BCx and ASTC are specifically targeting texture compression/decompression on GPUs and are designed to be random-access without entropy encoding. JPEG XL and AVIF are more traditional image compressors and include entropy encoding, which is a set of techniques that do not mesh well with the random access requirement for textures. We have included them still, since they are industry standards and because it may be worthwhile to investigate how our method fares against such advanced techniques. In fairness, it should be noted that neither JPEG XL nor AVIF were likely designed to reach bitrates as low as 0.2 BPPC, which is NTC's lowest target. We have also used Basis/KTX2 [30], which is part of the Khronos standard. This format also uses entropy encoding, but during decompression, it can transcode to many existing block-based texture compressions schemes, e.g., BCx, ETC, ASTC.

H.1 BCx Compression

For BCx compression [45], we performed a smaller investigation of existing tools, including AMD's Compressonator,[1] NVIDIA's Texture Tools,[2] and Intel's Fast ISPC Texture Compressor.[3] We used eight diffuse textures, eight normal maps, seven displacement maps, and seven roughness textures for this evaluation. The diffuse, normal, and displacement maps were from PolyHaven, and the roughness textures from ambientCG. The average PSNR for these three compressors over all textures were within ±0.2 dB. While the ISPC texture compressor had the highest average score, we could not find a command line tool version of it, and compressing a large number of high-resolution texture sets, including mipmaps, manually with their GUI was was prohibitively expensive.

For one-channel textures, e.g., roughness and displacement maps, AMD's tool wrote incorrect output files, so we always used NVIDIA's tool for those. For the diffuse textures and normal maps, AMD's tool produced slightly better result, so we used AMD's tool for those. The highest parameter setting was used for NVIDIA's tool, while we used two refine steps for BC1 (AMD) and quality 0.25 for BC7 (AMD). Going above those settings, mostly increased compression times but not quality.

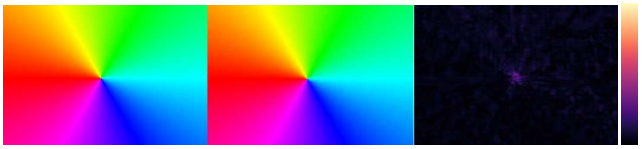

Fig. 19. A colorful and (presumably) difficult gradient texture compressed with NTC 0.2 does not show visible banding, color posterization, or discoloration. **Left:** Reference. **Middle:** Compressed. **Right:** ꟻLIP error image and corresponding color map.

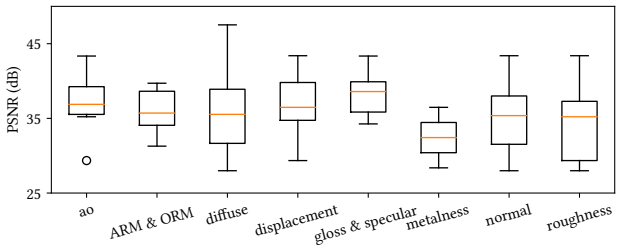

Fig. 20. Compression quality of different material properties for NTC 0.2. The orange lines show the median PSNR value for the respective texture types.

H.2 ASTC Compression

For ASTC [55], we used the texture tool from ARM, who developed ASTC,[4] with the `-exhaustive` flag, which provided best quality. Note that we used the two most aggressive variants of ASTC, which compressed 12×12 and 10×10 tiles. All variants store 16 bytes per tile, so using 12×12 tiles gives $128 \cdot 8/(12 \cdot 12) \approx 0.89$ bits per pixel for a three-channel texture. Furthermore, note that, in Figure 9 in the main paper, the ASTC results show average BPPC of around 0.5. This is a consequence of storing BPPC over an entire texture set, which may include both one- and three-channel textures. The same holds for other methods.

[1] https://gpuopen.com/compressonator/
[2] https://developer.nvidia.com/nvidia-texture-tools-exporter
[3] https://github.com/GameTechDev/ISPCTextureCompressor
[4] https://github.com/ARM-software/astc-encoder

Table 6. Storage cost of NTC textures based on the compression profile and texture resolution. The network parameter size depends on the compression profile, but is constant for different texture sizes. Storage cost is independent of the input channel count.

	NTC 0.2			NTC 0.5			NTC 1.0			NTC 2.25		
Resolution	2k×2k	4k×4k	8k×8k	2k×2k	4k×4k	8k×8k	2k×2k	4k×4k	8k×8k	2k×2k	4k×4k	8k×8k
NW (kB)	24	24	24	27	27	27	25	25	25	27	27	27
Grids (MB)	0.875	3.5	14.935	2.125	8.5	36.269	4.25	17.0	72.534	9.5	38.0	162.135
Total (MB)	0.899	3.524	14.959	2.152	8.527	36.296	4.275	17.025	72.559	9.527	38.027	162.162

Table 7. PSNR values with scalar quantization (SQ-8) and vector quantization (VQ-8, VQ-16) after optimization for 30k steps.

SQ-8	VQ-8	VQ-16
27.4 dB	27.28 dB	27.63 dB

H.3 JPEG XL

For JPEG XL [2], we used the reference implementation[5] and its precompiled executables (v0.8.0) from November 2022.[6] We started by performing lossless compression, and if that succeeded in reaching the target bitrate, our compression script exited. Otherwise, our script performed a binary search to find the quality setting that provided the sought-after bits per pixel per channel (BPPC). To reduce compression times, we did an early-out if the compression rate was within 2.5% of the target compression rate. We used the second highest value (8) for the effort parameter, since going to 9 (highest) provided little to no additional quality, but further increased compression times.

H.4 AVIF

For AVIF [13], we used precompiled executables using v0.11.1.[7] Similar to JPEG XL, our compression script for AVIF started by attempting to do lossless compression and exited if that reached the target compression rate. For all compression with AVIF, we used the "constant quality" `-a end-usage=q` flag, since this is common practice, and we also used quantization settings `-min 0 -max 63`. Next, our script performed a binary search on the `-a cq-level` quantization parameter *without* chroma subsampling. For very low bitrates, such 0.2 BPPC, this setting did not always reach the target. In those cases, our script continued with a new binary search with chroma subsampling enabled (`-yuv 420`). In the end, the file with the resulting bitrate closest to the target bitrate was selected. We used `-speed 3` since that resulted in reasonable compression times and going lower did not substantially improve image quality.

H.5 Basis/KTX2

For Basis,[8] we downloaded the code in early January 2023, and compiled it to use OpenCL for faster compression. The flags we use for compression are: `-opencl -ktx2 -uastc -uastc_rdo_l 1.0 -comp_level 6`, where 6 offers the best image quality and takes the longest to compress. All our results uses the file size of the output from the compressor. For image quality, however, we unpacked the basis file and had their decompressor transcode them to BC4 and BC7, respectively, since those are the formats that we compare to in the main paper.

[5]https://github.com/libjxl/libjxl
[6]https://artifacts.lucaversari.it/libjxl/libjxl/latest/
[7]https://github.com/AOMediaCodec/libavif
[8]https://github.com/BinomialLLC/basis_universal

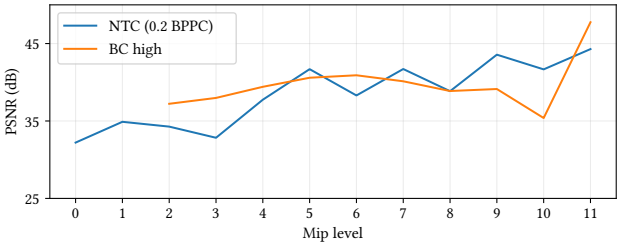

Fig. 21. Comparison of iso-storage compression quality at each mip level between our algorithm and the high-rate BC configuration, where we have used two fewer mipmap levels (0 and 1) to reach the same storage cost.

I ERROR AND QUALITY METRICS ON TEXTURE SETS AND MIP CHAINS

For reproducibility and future research, this section documents how our quality and error metrics were computed for texture sets and mip chains.

Given is a compressed texture set (including mip chains) $\mathbf{T} = \{\mathbf{T}_0^0, \mathbf{T}_0^1, \mathbf{T}_0^2, \ldots, \mathbf{T}_0^{M-1}, \ldots, \mathbf{T}_{N-1}^0, \mathbf{T}_{N-1}^1, \mathbf{T}_{N-1}^2, \ldots, \mathbf{T}_{N-1}^{M-1}\}$ with N textures and M mip levels, where mip j of texture i has resolution $w_j \times h_j \times c_i$. The values in the textures are assumed to be in $[0, 1]$. For each mip level, j, we concatenate the corresponding textures, creating a tensor $\mathbf{T}^j$ of shape $w_j \times h_j \times c$, where $c = \sum_i c_i$. The same concatenation is done for the reference texture set, $\mathbf{R}$, which contains the same number of textures as $\mathbf{T}$.

We compute the peak signal-to-noise ratio (PSNR) of $\mathbf{T}$ by summing the squared error over the mip chain and dividing by the total number of values in the tensor, yielding its mean squared error (MSE). That is, the PSNR is computed as

$$\mathrm{PSNR}(\mathbf{R}, \mathbf{T}) = -10 \log \left(\frac{\sum_j (\mathbf{R}^j - \mathbf{T}^j)^2}{c \sum_j w_j h_j} \right). \tag{1}$$

Computing LPIPS [88] for a texture set is a slightly more involved operation, as it requires 3-channel input. We consider a texture $\mathbf{T}_i^j$. If it only has a single channel, we repeat its channels to give a 3-channel texture, $\tilde{\mathbf{T}}_i^j$. If $\mathbf{T}_i^j$ has three channels, we let $\tilde{\mathbf{T}}_i^j = \mathbf{T}_i^j$. We then normalize the texture so that its values are in $[-1, 1]$, as is required by LPIPS, creating $\bar{\mathbf{T}} = 2\mathbf{T}_i^j - 1$. The same procedure is

used to create $\bar{\mathbf{R}}_i^j$. Next, we compute LPIPS between the two tensors $\bar{\mathbf{T}}_i^j$ and $\bar{\mathbf{R}}_i^j$. LPIPS yields an aggregate number $l_i^j = \text{LPIPS}\left(\hat{\mathbf{R}}_i^j, \hat{\mathbf{T}}_i^j\right)$ for the texture. We multiply the result by the number of channels c_i, effectively making a 3-channel texture worth three times as much as a single-channel texture when computing LPIPS over the texture set. Note that this is similar to the PSNR computations above. In addition, we multiply the result by the number of texels in the texture. We again sum the results over all levels of the mip chain and divide by the total number of values in the texture set. Formally, we have

$$\text{LPIPS}(\mathbf{R}, \mathbf{T}) = \frac{\sum_i \sum_j w_j h_j c_i l_i^j}{c \sum_j w_j h_j}. \tag{2}$$

We note that LPIPS was computed using the `net='alex'` LPIPS model, as proposed by the authors of the metric. Furthermore, LPIPS requires images that have resolutions of at least 32×32. For textures smaller than this, we apply zero-padding to make their sizes 32×32.

The structural similarity index (SSIM) [81] is a single-channel measure, providing a quality index between 0 and 1, where higher is better. To instead report errors, we compute 1−SSIM. We get an SSIM value for each channel in each texture in the mip chain. Similarly to the other two metrics, we weigh the result by the number of texels on the current mip level and normalize by the number of values in the texture set to get the final result. We get

$$1 - \text{SSIM}(\mathbf{R}, \mathbf{T}) = 1 - \frac{\sum_i \sum_j w_j h_j c_i \text{SSIM}\left(\mathbf{R}_i^j, \mathbf{T}_i^j\right)}{c \sum_j w_j h_j}, \tag{3}$$

where $\text{SSIM}\left(\mathbf{R}_i^j, \mathbf{T}_i^j\right)$ computes SSIM for each channel in the texture and returns the average. The SSIM computations include filtering with an 11×11 Gaussian kernel. We do not apply the kernel for images that are smaller than the kernel.

When we report LPIPS and SSIM errors over the entire data set, we take the mean of the errors computed for each texture set in the data set. For PSNR, we compute the average of the per-texture MSE values retrieved during the computation of the PSNR values. The aggregate PSNR is then computed using that average. Note that neither of these three aggregate error and quality values consider the resolution of the textures in the texture set nor the total number of channels present within it. The effect of this is that each texture in our diverse texture set has the same impact on the aggregate score. Furthermore, note that the computations in Equations 1-3 gives more weight to the lower levels (i.e., higher resolutions) of the mip pyramid compared to the higher. The reasoning behind this is that the lower levels will cover more pixels when used in rendered images.

Finally, we note that when we, for Figure 21, compute the average quality for mip level m over the entire data set, we aggregate over all available mips at that level, independently of their resolution. Consider the aggregate error value for the second mip level, for example. Despite the second mip level of a texture with resolution 1024×1024 having resolution 512×512 while the second mip level of a 4096×4096 texture has resolution 2048×2048, we add both of their error values into the aggregate for the second mip level, without weighting based on resolution.

J EVALUATION TEXTURE SET DETAILS

Figures 22-24 show the twenty texture sets included in our evaluation set. The sizes of the textures range from 2048×2048 to 8192×8192. The textures were either created by the authors or retrieved from in-house or external sources. The external sources were: ambientCG (https://ambientcg.com/), EISKO© (https://www.eisko.com/), KaiMoisch (https://kaimoisch.com/free-textures/), and PolyHaven (https://polyhaven.com/). Consisting of only a diffuse texture, the "gradient 4k" texture set contains the fewest number of channels (three). This texture set is the only one created by the authors and is provided as a difficult case. The "Louise 4k" set contains the most channels (12 in total, consisting of diffuse: 3, normal: 3, roughness: 1, subsurface: 1, ambient occlusion: 1, displacement: 1, gloss: 1, and specular: 1). Grayscale textures that were stored as RGB were converted to one-channel textures and constant textures were removed. EISKO and KaiMoisch provided 16-bit textures, which we converted to 8-bit before we used them.

The normal maps were originally provided as three-channel textures. We note that the BC5 format is a two-channel format targeting normal maps, under the assumption that the third component can be computed if the normals are of unit length. Intially, we planned to convert all normal maps to two channels, but found that the majority of the normal maps did not have normals of unit length. Since the length of the normal sometimes is used to store another property (for filtering normals, for example), we left them as three-channel textures to avoid changing the intent of the texture artists. However, since the quality of the normals is often extremely important, we used the higher-quality BC7 format (instead of BC1).

We create the mipmap chains with a high-quality Lanczos downsampling filter and stop when we have reached the level with a resolution of 4×4 pixels, since that is the tile size of most block-based GPU compression schemes.

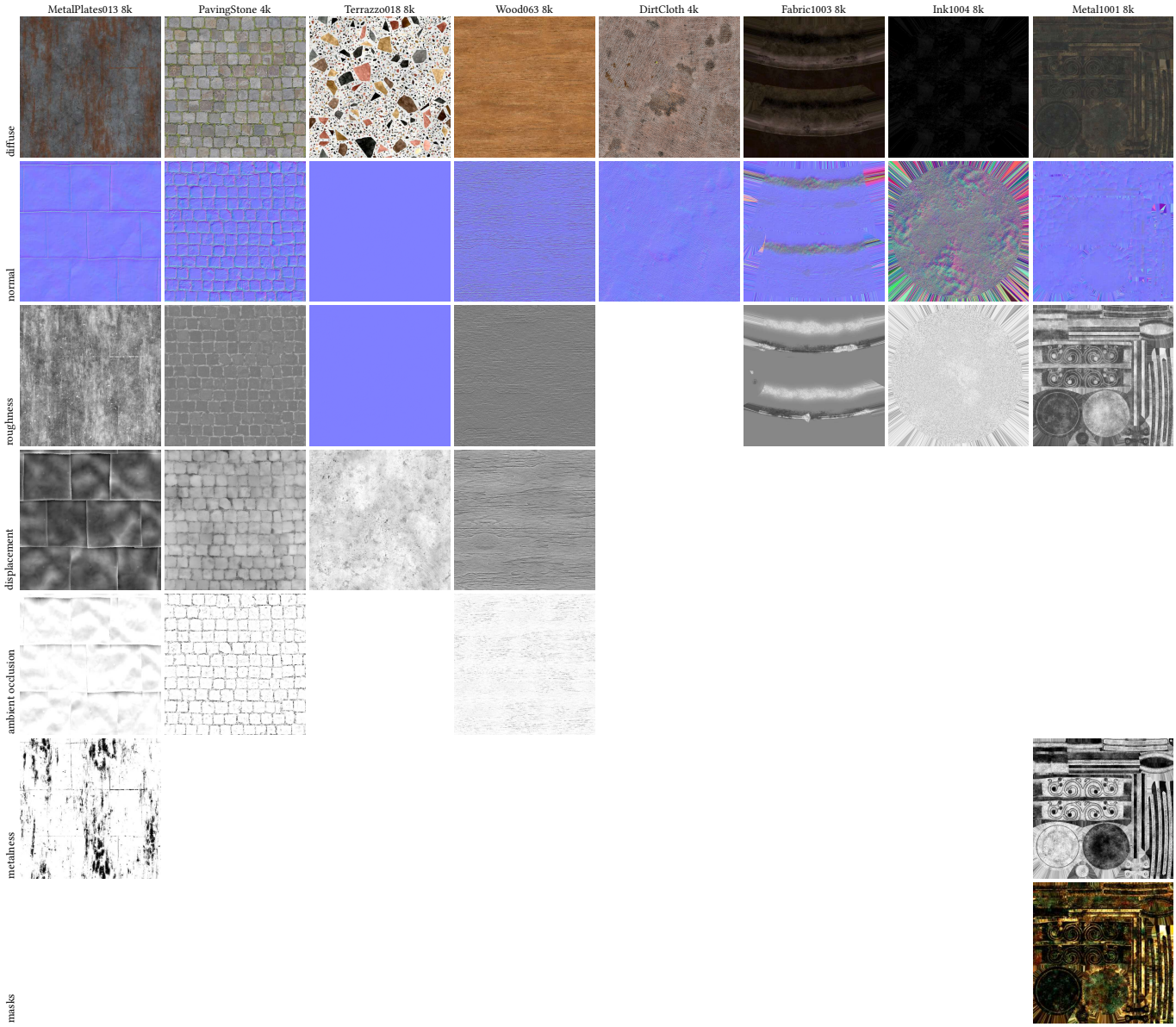

Fig. 22. Texture sets number 1.

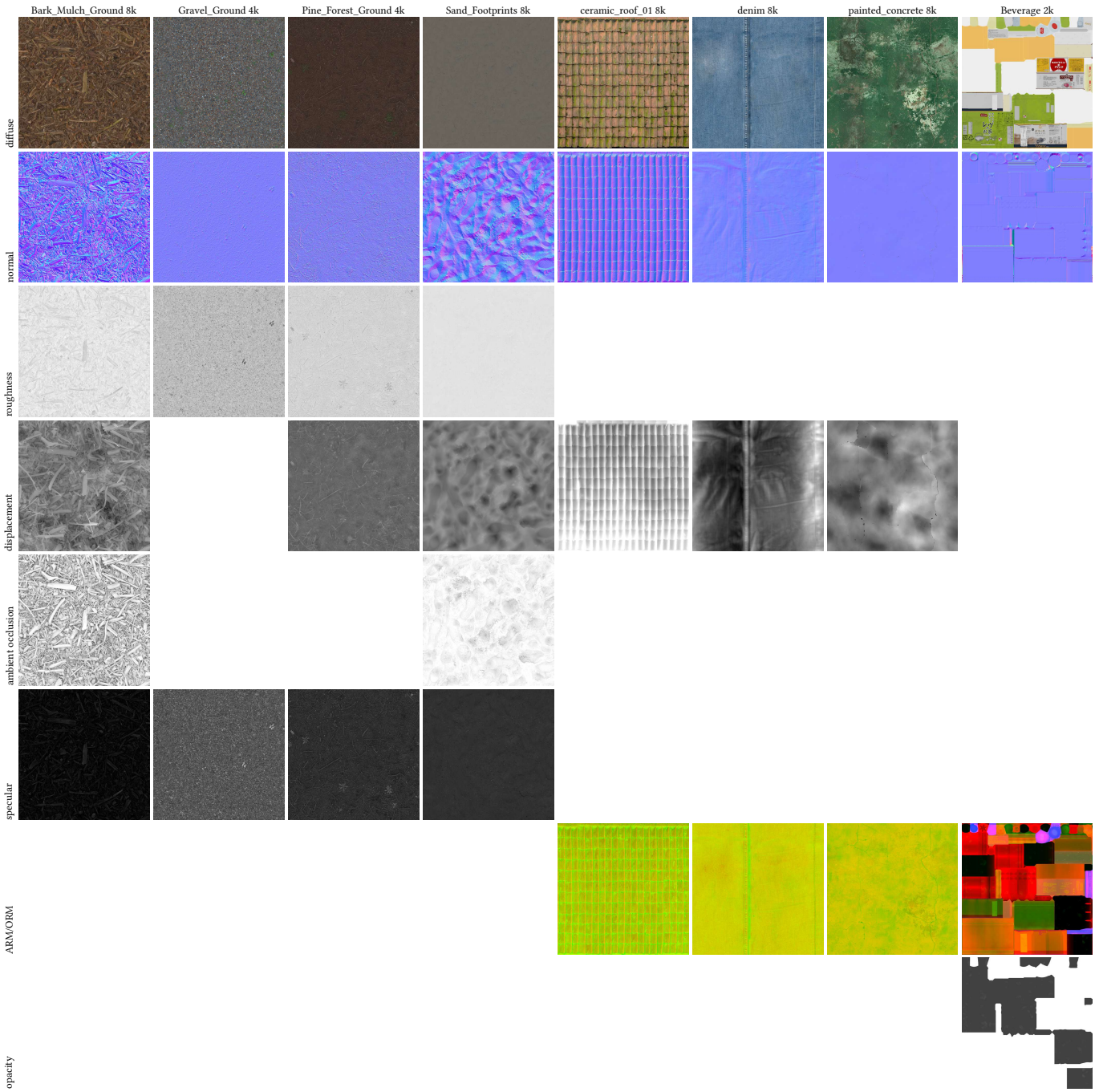

Fig. 23. Texture sets number 2.

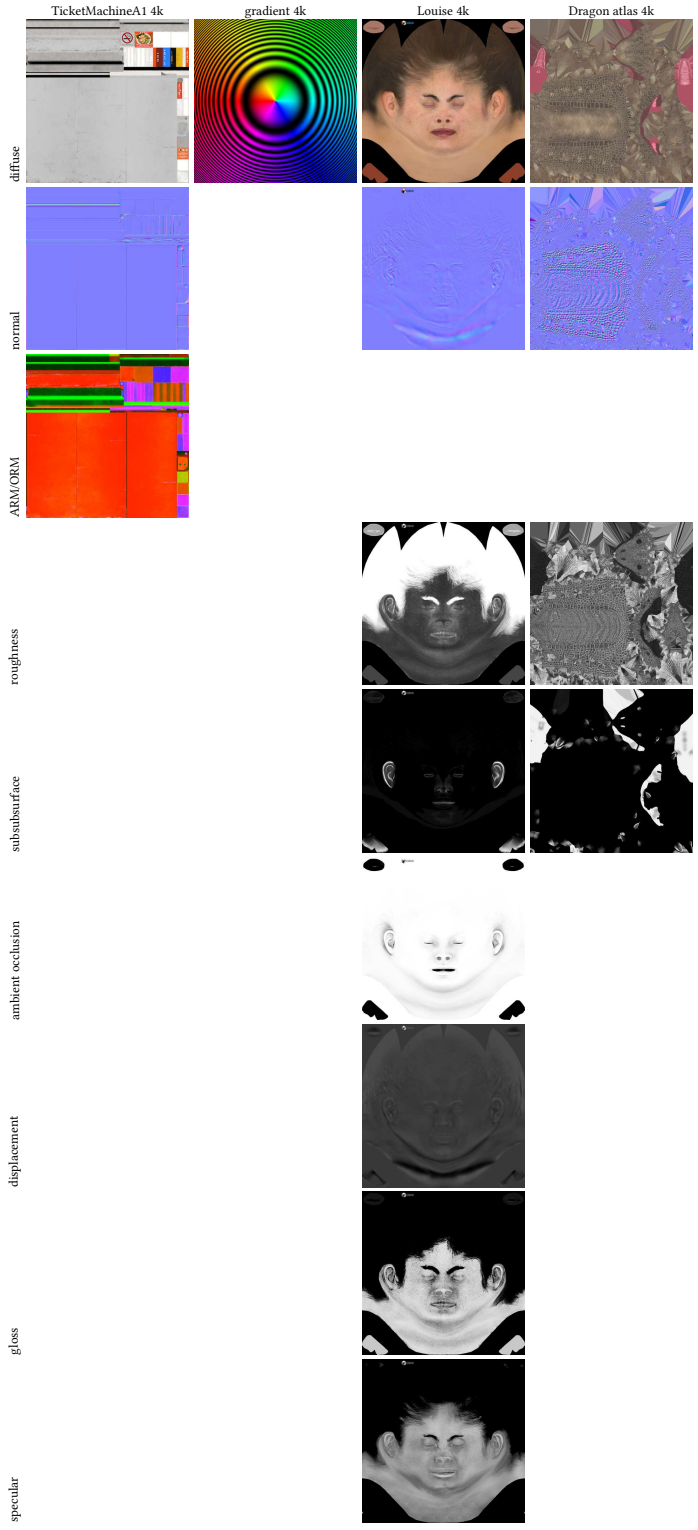

Fig. 24. Texture sets number 3. Animatable Digital Double of Louise by EISKO© (www.eisko.com).