High-Performance Graphics (2025)
A. Knoll and C. Peters (Editors)

# Hardware Accelerated Neural Block Texture Compression with Cooperative Vectors

L. Belcour*,[1] and A. Benyoub*,[1]

[1]Intel Labs, France

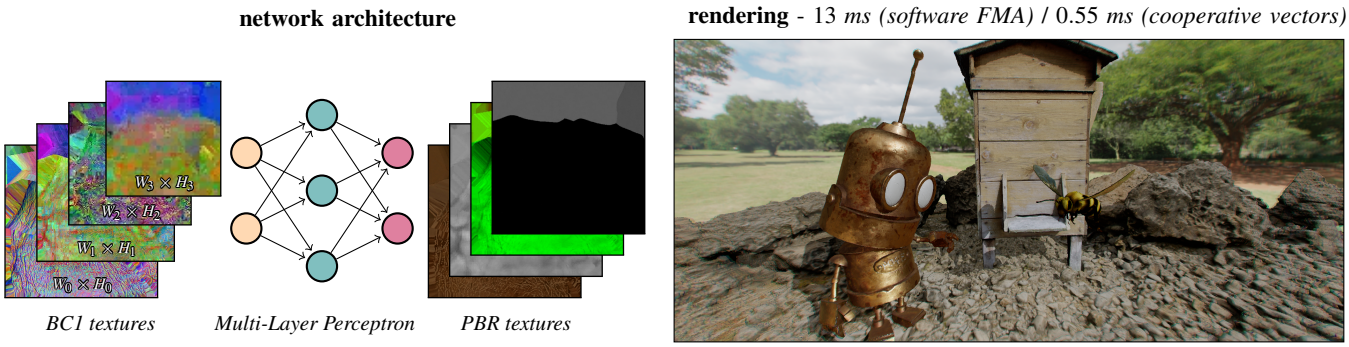


**Figure 1:** ***Rendering physically-based materials using neural block texture compression.*** *We compress PBR texture sets using block compressed latent images decoded by a multi-layer perceptron (left). We show how to accelerate this architecture using the cooperative vectors extension to obtain up to* 23× *speedup compared to a compute based with fused multiply add (FMA) implementation (right).*

**Abstract**

*In this work, we present an extension to the neural texture compression method of Weinreich and colleagues [WDOHN24]. Like them, we leverage existing block compression methods which permit to use hardware texture filtering to store a neural representation of physically-based rendering (PBR) texture sets (including albedo, normal maps, roughness, etc.). However, we show that low dynamic range block compression formats still make the solution viable. Thanks to this, we show that we can achieve higher compression ratio or higher quality at fixed compression ratio. We improve performance at runtime using a tile based rendering architecture that leverage hardware matrix multiplication engine. Thanks to all this, we render 4k textures sets (9 channels per asset) with anisotropic filtering at 1080p using only* 28*MB of VRAM per texture set at* 0.55*ms on an Intel B580.*

## 1. Introduction

Modern games rely increasingly on high-resolution textures (often $4096 \times 4096$ pixels) to store the appearance of assets. Since physically-based shading models require many parameters (albedo, normal map, roughness, metalness, *etc*) [HMB*20] for their evaluation, many textures have to be stored for a single asset. This comes at the cost of storage capacity both on disk and on graphics memory (VRAM). In fact, some AAA games can go beyond 100Gb [Fow23] on disk, with most of this storage dedicated to textures. To mitigate this, texture compression algorithm must be used. However, decompression will impact the performance either at runtime or during loading time. Furthermore, decompression must be compatible with texture mipmapping for level-of-details. Hence, hardware-accelerated compression methods such as block compression [Khr25] (BC1 to BC7) are the *de facto* standard. Still, this type of compression does not use the fact that different textures are used for the same asset and could be further compressed by leveraging the correlation between different parameters.

**Neural Material Compression.** Higher compression ratio through texture grouping is the key idea behind neural material compression. Instead of compressing textures individually, neural material compression methods [FHL*24, VSW*23, WDOHN24] stack together all textures belonging to the same asset (called a texture set) and compress this tensor into a lighter representation. A simple Multi-Layer Perceptron is used to decode this representation to an approximation of the original textures. Still, those methods trade lower footprint on disk and/or VRAM for compute since a neural network has to be evaluated.

* Authors share equal contribution to this work



DOI: 10.2312/hpg.20251173

**Our method.** Our work aims to improve the Block Compressed Features (BCF) method [WDOHN24] both in terms of compression ratio, quality, as well as performance. More specifically, we

- use BC1 in place of BC6 with a different resolution layout, allowing more compression or quality depending on the requirements (see Section 3).
- show that anisotropic filtering is possible with those models (see Section 3.3).
- reduce decoding time with hardware acceleration of matrix-vector multiplication made efficient thanks to a tile-based rendering architecture (leveraging the *cooperative vector* extension) (see Section 4).

## 2. Previous Work

Texture compression plays a crucial role in optimizing GPU memory usage and performance in modern graphics applications. Over the years, significant advancements have been made to improve compression efficiency, decoding speed, and visual quality while keeping memory requirements manageable (see the evolution of JPEG formats [Wal92, CES00, AVAB*19] for example). However, not all compression formats are usable in 3D graphics, as hardware accelerated decoding and filtering must be available.

**Texture Compression.** The block Compression (BC) family of formats [DM79, FNK94], (BC1 to BC7), is widely used in gaming and graphics due to its balance between compression ratio, visual quality, and decoding speed. These formats use fixed-size blocks (typically 4x4 pixels) and are optimized for GPU hardware, enabling efficient decompression during rendering.

ASTC [NLP*12] is a more flexible texture compression standard with adaptive range of block sizes, from $4\times4$ to $12\times12$ pixels, allowing for a finer control over the compression rate and visual quality. Despite its advantages, BC formats are often prefered over ASTC as they are widely supported.

**Neural Image Compression.** Many work have leveraged the use of Machine Learning for image compression [BLS17]. However, most of these works are not usable in a real-time context, where fast decoding speed is mandatory.

Instead some works have focused on this specific challenge of compressing images with real-time usage. For example, Datta et al. [DMD*23] learn indirections to a buffer of values. Although it achieves good compression ratios, dependent fetches can be harmful to performance. Fujieda and Harada [FH24] introduced Neural Texture Block Compression (NTBC), which uses neural networks to map uncompressed textures to compressed texture blocks. Their approach reduces disk storage costs by approximately 70% while maintaining real-time performance. However, they decompress textures to BC format in VRAM.

**Neural PBR Material Compression.** Instead of directly compressing images, some work leverages the correlation between the physically based materials channels. This can lead to higher compression ratios. Most notably, [VSW*23, FHL*24] use multiple grids and positional encoding to decode pixels. Those methods target high compression ratios, but require higher compute (2 hidden layers with 64 hidden channels) than alternatives.

Weinreich et al. [WDOHN24] introduced a method that stores learned latent features as block-compressed high-dynamic range BC6 textures. Their approach leverages a small MLP decoder (1 hidden layer) that runs on the filtered latent features to infer the texture data. This allows for real-time decompression directly within shaders. We build on their method and show how to improve compression ratio and inference speed using LDR block compression and hardware accelerated inference on tiles.

## 3. Neural Block Texture Compression

In the following, we first describe our method's neural architecture (Section 3.1). Then, we detail how textures are compressed in our prototype (Section 3.2). We also discuss runtime evaluation and texture filtering (Section 3.3).

### 3.1. Network Architecture

Figure 1 displays the architecture of our model: a set of block compressed textures with different resolutions stores a latent representation. To evaluate a specific pixel at a given UV coordinate $p(\mathbf{uv})$, we evaluate each latent texture $\mathbf{c}_k = T_k(\mathbf{uv})$, concatenate all latent colors $\mathbf{x} = [\mathbf{c}_0 \ \dots \ \mathbf{c}_K]$ and feed the resulting vector to a Multi-Layers Perceptron (MLP) to obtain a vector of features (albedo, normal, ...): $\mathbf{f} = f(\mathbf{x})$.

**Latent BC1 compression.** Each latent texture $T_k$ has its own resolution $(W_k \times H_k)$ and is stored in BC1 format (see Khronos documentation [Khr25] for details). Hence, each pixel is defined as the interpolation of two endpoints $\mathbf{e}_0, \mathbf{e}_1$ with a blending factor $\alpha$:

$$T(\mathbf{uv}) = (1-\alpha(\mathbf{uv}))\,\mathbf{e}_0(\mathbf{uv}) + \alpha(\mathbf{uv})\mathbf{e}_1(\mathbf{uv}).$$

With BC1 format, the endpoints are shared by groups of $4\times4$ texels while every texel has its own $\alpha$. In practice, $\alpha$ is quantized using 2 bits and $\mathbf{e}_0$, $\mathbf{e}_1$ are quantized with a $[5,6,5]$ bits pattern.

**Difference with BCF.** A key difference with the method of Weinreich *et al.* [WDOHN24] is that they use BC6 compression. Their main reason to use BC6 is that it allows to decode HDR values. However, we did not find it to be necessary in practice. Instead, we can store twice the amount of BC1 textures for the same footprint in BC6. In the following, we refer to BCF [WDOHN24] as BCF6 and ours as BCF1*.

**Variants.** We tested two variants with four textures per material: VARA and VARB. They differ by the resolution of the latent texture: we designed VARA to have almost the same compression ratio as BCF6 while VARB almost double this compression ratio.

| variant | resolutions | | | |
|---|---|---|---|---|
| VARA | $W\times H$ | $W\times H$ | $\frac{W}{2}\times\frac{H}{2}$ | $\frac{W}{2}\times\frac{H}{2}$ |
| VARB | $W\times H$ | $\frac{W}{2}\times\frac{H}{2}$ | $\frac{W}{4}\times\frac{H}{4}$ | $\frac{W}{8}\times\frac{H}{8}$ |

* we could use other kind of image compression such as BC2, BC3, *etc*. We did not test that in this article.

Furthermore, we apply sub-pixel shifts during the evaluation of the latent textures. We shift by half a texel the second and fourth textures. This avoids the alignment of latent blocks to reduce artefacts . We show in Figure 2 the impact of both variants on image quality. See our supplemental material for more results.

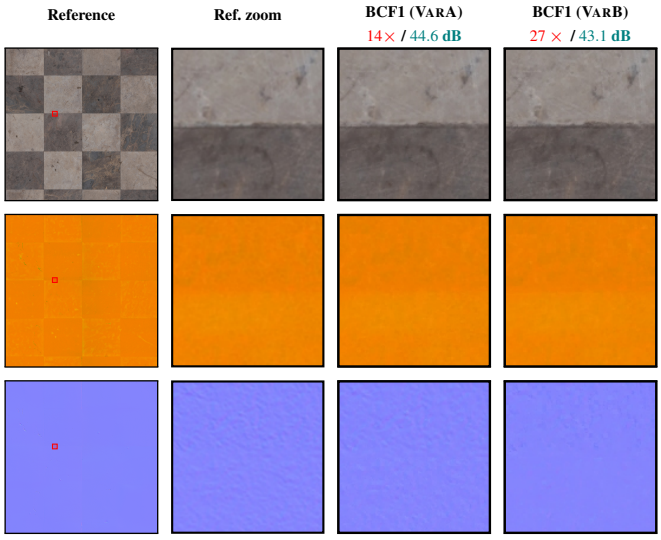


**Figure 2:** ***Quality of the different variants.*** *Using* VARA *or* VARB *permit to adapt the compression level (in red) with an impact on quality (average PSNR in green). As we double the compression ratio, the block artifacts of BC1 reduce the quality of the inference.*

### 3.2. Training

To compress texture into our latent BC1 format, we replicated a BC1 texture unit in PyTorch [PGM*19]. Since BC1 texture store mip hierarchies as separate BC1 textures, we optimize them as well. During training, we sample a LOD level and regress the associated texture. We sample the LOD level similarly to Vaidyanathan [VSW*23].

We apply a L1 loss between the decoded features and reference features sampled using a trilinear filtering. Gradients are passed through the MLP and each element of the BC1 textures are updated using the Adam optimizer (with $lr = 10^{-3}$ and $lr = 10^{-2}$ respectively). For each BC1 texture, we store its endpoints and alpha as floating point tensor. To decompress the unsigned quantized values ($\bar{\alpha}$, $\bar{\mathbf{e}}_0$, and $\bar{\mathbf{e}}_1$), we apply a sigmoid activation function to the floating point values before performing quantization aware training:

$$\bar{\alpha} = \text{quant}\left(\text{sigmoid}\left(\alpha\right), [2]\right) \quad (1)$$

$$\bar{\mathbf{e}}_0 = \text{quant}\left(\text{sigmoid}\left(\alpha\right), [5,6,5]\right) \quad (2)$$

$$\bar{\mathbf{e}}_1 = \text{quant}\left(\text{sigmoid}\left(\alpha\right), [5,6,5]\right) \quad (3)$$

where $\text{quant}(x,b)$ evaluate the quantized form of $x$ using $b$ bits but let the gradients of its floating point value pass through.

We did not find evidence that optimizing first unquantized latent maps before quantizing during training was brining any gain in quality. Hence, we optimize our model with quantized representation from the start.

### 3.3. Anisotropic Filtering

We found that using such optimization provides good results with anisotropic filtering when the latent texture are fetched with non-isotropic kernels. While our models are never trained on anisotropicaly filtered data, they still provide visualy correct anisotropic filtering without any change. We hypothetize that since the network has learned to infer the behaviour of bilinearly filtered latents, as a side effect it learned that blending latent codes results in blended outputs. Since anistropic filtering is a blend of many taps, this behavior is not surprizing. In Figure 3, we compare renderings of our decompressed textures with respect to the reference using isotropic or anisotropic filtering for both.

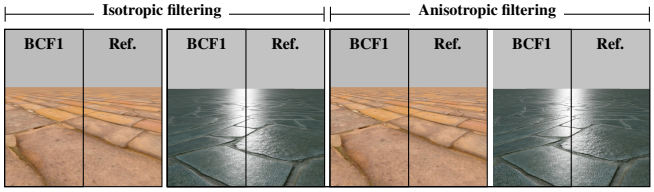


**Figure 3:** ***Isotropic and anisotropic fitlering.*** *We validate that training with isotropic filtering permits to also extrapolate anisotropic filtering in our* VARA.

## 4. Runtime & Hardware Acceleration

Given that the neural texture set is stored in the BC1 format and leverages the texture unit, the main bottleneck in the runtime evaluation lies in how the MLP is evaluated and where in the graphics pipeline it is evaluated.

**Rendering architecture.** In our prototype, we implemented a rendering pipeline built around a *Visibility buffer* [BH13]. This does mimick the integration of our method in a modern game engine. The *Visibility pass* is followed by a *Classification pass* that sort pixels requiring neural inference. Then, a *G-Buffer pass* consumes the BCF1 texture set and produces all the data required for shading (albedo, normal, roughness, *etc*). Finally, a *Lighting pass* consumes the GBuffer and shades each pixel. We implemented a simple shading model that blends a diffuse lobe and a GGX lobe using prefiltered Image Based Lighting and a directional light [LdR14].

**Neural inference in G-Buffer pass.** We use the texture unit for an efficient fetch of block compressed latent features and unless noted, we use hardware accelerated anisotropic filtering. To improve performance in the *G-Buffer pass*, we take advantage of the hardware matrix multiplication engines to accelerate the inference of the differents layers of the Multi-Layer Perceptron (MLP). Our implementation use the `cooperative vectors` extension [Jef24,cas25]. This provides better performance than using the `cooperative matrix` API or using software FMA for matrix multiplication. In our use of the cooperative vectors, we only use the column major layout and let the optimized opaque layout for future tests. But for *cooperative vectors* to be efficient, threads in the same workgroup must use the same base matrix. Hence, we need to group together pixels using the same MLP.

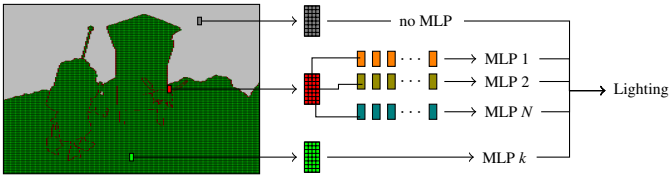


**Figure 4:** ***Tile-based classification.*** *We tile the output of the Visibility pass in $8 \times 4$ pixels tiles. Each tile is classified as either non neural (gray), neural with a single MLP (green), and neural with mixed MLPs (red). To avoid divergence, we repack all mixed tiles to have $8 \times 4$ groups of pixels with the same MLP (orange, marron, blue). Once this sorting is done, we can perform our Lighting pass.*

**Tile-based classification.** For improved performance, we leverage a tile-based architecture where both the *G-Buffer* and *Lighting* passes operate on $8 \times 4$ pixels tiles to match workgroups size. During a *Classification pass*, we classify each $8 \times 4$ tile based on its content so as to perform neural decompression only on tiles that have at least one pixel using a neural texture set (see Figure 4). We detail this in Algorithm 1. We first classify all tiles into three types (`classificationA` in Algorithm 1): *no neural* for tiles that do not require neural decompression; *simple neural* for tiles with the same network for all its pixels; and *mixed neural* for tiles with at least two different networks. We then perform an additional compute pass on *mixed neural* tiles and sort their pixels by their MLP index (`classificationB` in Algorithm 1). This groups pixels with the same MLP in $8 \times 4$ tiles on which we execute the *Lighting pass* before splatting the result in the associated pixel.

## 5. Results

We tested both the quality and performance of our method. We refer to our supplemental material and video for more results. In all cases, we compressed PBR textures using a PyTorch implementation of the compressor.

**Quality and compression.** We tested the quality and compression factor of our method using a dataset of texture sets from Polyhaven [Z*]. For every texture set, we selected the diffuse, normal, roughness, metalness and ambient occlusion channels, resulting in texture sets with 9 channels. We used $4096 \times 4096$ pixels textures in every cases.

We analyzed the quality (using the PSNR in decibels) for our different variants of BCF1. In Table 1, we report the different configurations we tested as well as the min, max, and average PSNR (in decibel). We found that using wider MLP is beneficial for quality. In the following, unless noted, we report results for $D = 64$.

We report as well the quality w/r/t the compression ratio of our variants and compare them to Ubisoft's BCF6 and Nvidia's NTC (1 and $1/2$ bits per channel per pixel variants). Both BCF6 and NTC are our implementation, following the respective articles. For BCF6, we used the same mode for all blocks (mode 10 with RGB encoded with 6 bits per channel). We report the average PSNR and compression ratio in Figure 5. Compared to BCF6, our BCF1 varA method produce similar quality with a small gain in compression.

| variant | L layers | hidden dim | PSNR |
|---|---|---|---|
| **varA** | $L = 1$ | $D = 16$ | 22.22 / 40.37 / 49.67 |
| | $L = 1$ | $D = 32$ | 22.70 / 40.85 / **50.16** |
| | $L = 1$ | $D = 64$ | **23.34** / **40.91** / 50.02 |
| **varB** | $L = 1$ | $D = 16$ | 22.11 / 38.28 / 48.74 |
| | $L = 1$ | $D = 32$ | 22.34 / 38.75 / 49.11 |
| | $L = 1$ | $D = 64$ | **22.91** / **39.02** / **49.18** |

**Table 1:** ***Dimension of the MLP.*** *We compressed textures from the Polyhaven texture dataset and report the quality in PSNR for different configurations of our variants.*

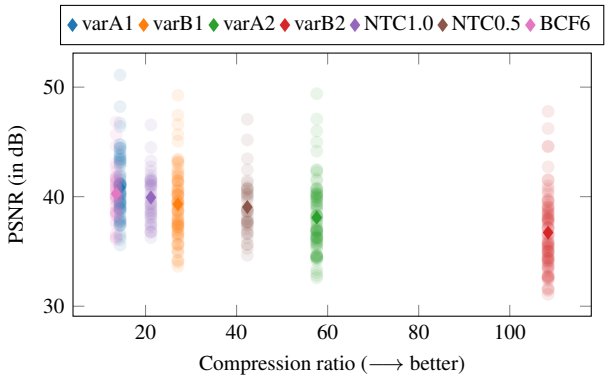


**Figure 5:** ***Quality and compression ratio.*** *We compare the quality of our variants, varA and varB with latents at full resolution (varA1, varB1) and half resolution (varA2, varB2) respectively. We also plot the quality obtain using Nvidia's NTC (both 0.5 and 1.0 variants) and BCF using BC6H compression.*

**Performance.** We implemented our method using the DirectX 12 API following Section 4. Figure 1 shows our prototype with a scene composed of 4 assets (a robot, a bee, a bee hive, and rocks) each with its own neural material at $4096 \times 4096$ resolution. We tested our implementation on an Intel B580 and an Nvidia RTX 4090 GPUs. We set the screen resolution at $1920 \times 1080$ pixels for every result. Please refere to our supplemental video for more results.

We report performance measurements in Table 2. For every result, we skip the *Visiblity* and *Lighting pass* from the timings to focus on the impact of materials evaluation. We compare the impact of computing matrix multiplication in software using FMA versus leveraging the hardware acceleration using the cooperative vector API. When using wider MLPs ($D >= 32$), using cooperative vectors permits up to a $\times 40$ speedup.

## 6. Limitations

**Scaling issue** Our classification permits to handle different number of MLPs but is not yet optimal. We currently use global atomics to affect pixels in mixed tile to uniform tiles. This works fine when a small percentage of tiles are mixed, but shared memory's atomics would be more suited if all the screen is filled with mixed tiles.

**Sharing of MLP.** One possibility is to share a unique MLP for different materials, hence reducing the number of mixed tiles. This would improve performances at the expense of quality. We leave this feature for future works.

| | | no neural | software (FMA) | coop vectors | PSNR |
|---|---|---|---|---|---|
| **varA** | $D = 16$ | 0.035 / 0.11 *ms* | 0.085 / 0.183 *ms* | 0.085 / 0.077 *ms* ($\times 1-2$) | 34.70 / 37.49 / 38.52 |
| | $D = 32$ | | 0.785 / 0.388 *ms* | 0.07 / 0.078 *ms* ($\times 5-11$) | 38.62 / 41.20 / 43.82 |
| | $D = 64$ | | 2.8 / 2.3 *ms* | 0.06 / 0.113 *ms* ($\times 20-46$) | **38.96** / **41.76** / **44.43** |
| **varB** | $D = 16$ | 0.035 / 0.11 *ms* | 0.08 / 0.183 *ms* | 0.085 / 0.075 *ms* ($\times 1-2$) | 34.97 / 35.83 / 36.83 |
| | $D = 32$ | | 0.6 / 0.387 *ms* | 0.06 / 0.079 *ms* ($\times 5-10$) | **36.87** / 39.55 / **43.22** |
| | $D = 64$ | | 2.7 / 2.3 *ms* | 0.055 / 0.11 *ms* ($\times 20-49$) | 36.86 / **39.86** / 43.20 |

**Table 2:** ***Performance and quality of our variants.*** *We measured timings of our prototype with (coop vectors) and without (software FMA) using hardware matrix multiplication on the Polyhaven assets database. Depending on the dimension of the MLP, hardware acceleration can provide a benefit. We also report the min / average / max PSNR for each variant. We measured timings on an Intel B580 (in blue) as well as an Nvidia RTX 4090 (in green) GPUs.*

## 7. Conclusion

We have shown that using hardware acceleration provides a substancial benefit for the evaluation of neural materials. Furthermore, using *low dynamic range* latent representations (using BC1) permits to obtain similar quality than previous work using *high dynamic range* storage at the same compression rate and attain higher compression rate with a reasonnable quality drop.

We have shown that our compression approach when used in conjunction with coop vectors presents a viable alternative to traditional compression. We believe that standardizing a neural format will make neural texture compression even more appealing and thus constitutes a valuable future work.

## References


[AVAB*19] ALAKUIJALA J., VAN ASSELDONK R., BOUKORTT S., BRUSE M., COMȘA I.-M., FIRSCHING M., FISCHBACHER T., KLIUCHNIKOV E., GOMEZ S., OBRYK R., ET AL.: Jpeg xl next-generation image compression architecture and coding tools. In *Applications of digital image processing XLII* (2019), vol. 11137, SPIE, pp. 112–124. 2

[BH13] BURNS C. A., HUNT W. A.: The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (August 2013), 55–69. URL: http://jcgt.org/published/0002/02/04/. 3

[BLS17] BALLÉ J., LAPARRA V., SIMONCELLI E. P.: End-to-end optimized image compression. In *International Conference on Learning Representations* (2017). URL: https://openreview.net/forum?id=rJxdQ3jeg. 2

[cas25] Enabling neural rendering in directx: Cooperative vector support coming soon, 2025. URL: https://devblogs.microsoft.com/directx/enabling-neural-rendering-in-directx-cooperative-vector-support-coming-soon/. 3

[CES00] CHRISTOPOULOS C. A., EBRAHIMI T., SKODRAS A. N.: Jpeg2000: the new still picture compression standard. In *Proceedings of the 2000 ACM workshops on Multimedia* (2000), pp. 45–49. 2

[DM79] DELP E., MITCHELL O.: Image compression using block truncation coding. *IEEE transactions on Communications 27*, 9 (1979), 1335–1342. 2

[DMD*23] DATTA S., MARSHALL C., DONG Z., LI Z., NOWROUZEZAHRAI D.: Efficient graphics representation with differentiable indirection. In *SIGGRAPH Asia 2023 Conference Papers* (2023), pp. 1–10. 2

[FH24] FUJIEDA S., HARADA T.: Neural texture block compression. *arXiv preprint arXiv:2407.09543* (2024). 2

[FHL*24] FARHADZADEH F., HOU Q., LE H., SAID A., RAUWENDAAL R., BOURD A., PORIKLI F.: Neural graphics texture compression supporting random access. In *European Conference on Computer Vision* (2024), Springer, pp. 412–429. 1, 2

[FNK94] FRÄNTI P., NEVALAINEN O., KAUKORANTA T.: Compression of digital images by block truncation coding: a survey. *The Computer Journal 37*, 4 (1994), 308–332. 2

[Fow23] FOWLER C.: Extending in-game textures using cdns for 'call of duty: Modern warfare 2'. GDC'23, 2023. 1

[HMB*20] HILL S., MCAULEY S., BELCOUR L., EARL W., HARRYSSON N., HILLAIRE S., HOFFMAN N., KERLEY L., PATRY J., PIEKÉ R., ET AL.: Physically based shading in theory and practice. In *ACM SIGGRAPH 2020 Courses*. 2020, pp. 1–12. 1

[Jef24] JEFF B.: Vk_nv_cooperative_vector, 2024. URL: https://registry.khronos.org/vulkan/specs/latest/man/html/VK_NV_cooperative_vector.html. 3

[Khr25] KHRONOS: Vulkan documentation: Compressed image formats, 2025. URL: https://docs.vulkan.org/spec/latest/appendices/compressedtex.html. 1, 2

[LdR14] LAGARDE S., DE ROUSIERS C.: Moving frostbite to physically based rendering. In *SIGGRAPH 2014 Courses*. ACM, 2014. 3

[NLP*12] NYSTAD J., LASSEN A., POMIANOWSKI A., ELLIS S., OLSON T.: Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics Conference on High-Performance Graphics* (2012), pp. 105–114. 2

[PGM*19] PASZKE A., GROSS S., MASSA F., LERER A., BRADBURY J., CHANAN G., KILLEEN T., LIN Z., GIMELSHEIN N., ANTIGA L., DESMAISON A., KÖPF A., YANG E., DEVITO Z., RAISON M., TEJANI A., CHILAMKURTHY S., STEINER B., FANG L., BAI J., CHINTALA S.: *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019. 3

[VSW*23] VAIDYANATHAN K., SALVI M., WRONSKI B., AKENINE-MÖLLER T., EBELIN P., LEFOHN A.: Random-access neural compression of material textures. *arXiv preprint arXiv:2305.17105* (2023). 1, 2, 3

[Wal92] WALLACE G. K.: The jpeg still picture compression standard. *IEEE transactions on consumer electronics 38*, 1 (1992), xviii–xxxiv. 2

[WDOHN24] WEINREICH C., DE OLIVEIRA L., HOUDARD A., NADER G.: Real-time neural materials using block-compressed features. In *Computer Graphics Forum* (2024), vol. 43, Wiley Online Library, p. e15013. 1, 2

[Z*] ZAAL G., ET AL.:. URL: https://polyhaven.com/. 4

```
[numthreads(8, 4, 1)]
void ClassificationA(uint2 groupID: SV_GroupID, uint2 pixelCoords : SV_DispatchThreadID)
{
    // Get the material ID of the pixel
    uint32_t matID = visibility_buffer_mat_id(pixelCoords)
    if (valid_mat_id(matID))
    {
        // First we need to find if there are multiple MLPs within this work group
        uint32_t maxID = WaveActiveMax(matID);
        uint32_t minID = WaveActiveMin(matID);

        // How many pixels valid within the WG?
        uint validPixelsCount = WaveActiveSum(1);

        // Flatten the work group ID
        uint wgID = uint(groupID.x + groupID.y * _TileSize.x);

        // Append in the right lists
        if (WaveIsFirstLane())
        {
            // This workgroup has only one material and at least 16 active pixels
            if (maxID == minID && validPixelsCount > 16)
                _UniformTileAppendBuffer.Append(wgID);
            else
                _ComplexTileAppendBuffer.Append(wgID);
            _ActiveTileAppendBuffer.Append(wgID);
        }
    }
}

[numthreads(8, 4, 1)]
void ClassificationB(uint2 groupThreadID : SV_GroupThreadID)
{
    // Get the actual work group Index
    uint workGroupID = _ComplexTileConsumeBuffer.Consume();

    // Get it as coordinates
    uint wgX = workGroupID % _TileSize.x;
    uint wgY = workGroupID / _TileSize.x;

    // Compute the pixel coords
    uint2 pixelCoords = uint2(wgX * 8 + groupThreadID.x, wgY * 4 + groupThreadID.y);

    // Is this a valid pixel? If yes it needs to register
    uint32_t matID = visibility_buffer_mat_id(pixelCoords)
    if (valid_mat_id(matID))
    {
        // This is a mixed tile, so book a slot for this pixel in the right tile group
        uint targetSlot;
        InterlockedAdd(_MLPUsageBufferRW[matID], 1, targetSlot);

        // Get the the group offset
        uint32_t mlpGroupOffset = _MLPUsageBufferRW[UNIQUE_MLP_COUNT + matID];
        _IndexedTilesBufferRW[tileGroupOffset * WORK_GROUP_SIZE + targetSlot] = pixelCoords.x +
    pixelCoords.y * _ScreenSize.x;
    }
}
```

**Algorithm 1: Tile-based classification.** Our classification shader (here, in HLSL) use two passes: `classificationA` and `classificationB` to sort pixels into coherent tiles.