Towards Comprehensive Neural Materials: Dynamic Structure-Preserving Synthesis with Accurate Silhouette at Instant Inference Speed

Supplemental Material

1 IMPLEMENTATION

1.1 Quantization Aware Training

We implement the network and training in PyTorch [Paszke et al. 2019]. We leverage the Brevitas [Pappalardo 2023] library for neural network quantization, which provides automatic quantization aware training operations and is easy to export network parameters. We use symmetric, 8-bit per-tensor signed int quantization for both weight and activations (neural layer's input), and we do not use bias terms in our network. The benefit is clear: we only need two scaling parameters for each layer: one for the activation and one for the weight. Our three feature planes are not quantized, and values in them are still in FP32 to better support interpolation. After training, the network weights and feature planes are exported as binary files, and the scaling factors are hard-coded in our renderer.

1.2 BTF & Synthetic Data

We have trained two measured BTFs (LEATHER04 in Fig. 7 and LEATHER11[1] in Fig. 1, 8, and 12) from UBO2014 [Weinmann et al. 2014] dataset. Both of them contain 22,801 images at a size of 400×400.

Apart from the two measured BTFs mentioned above, the rest of the material examples we show in our paper and the videos are our own synthetic materials. To create these synthetic materials, we use a point set (10×10) that is uniformly distributed in the unit square to get the $\boldsymbol{\omega_o}$ and $\boldsymbol{\omega_i}$ with Shirley's concentric hemisphere parametrization [Shirley 2011; Shirley and Chiu 1997] respectively. The spatial resolution for the synthetic material is 800×800. We record the reflectance with the given $(\mathbf{u}, \boldsymbol{\omega_o}, \boldsymbol{\omega_i})$ in Nvidia's Falcor renderer [Benty et al. 2020] with its Standard Material model (microfacet-based). Consequently, the dataset for each synthetic material contains = 10,000 $(10 \times 10 \times 10 \times 10)$ images with a resolution of 800×800.

1.3 Training Details

For the optimizer, we apply AdamW and follow the same setting as TP [Xu et al. 2024]. In Table 1, we summarize the size of each feature plane. For feature plane sampling, instead of directly using theta and phi as texture coordinates as in Triple Plane [Xu et al. 2024], we use Shirley's concentric hemisphere parametrization [Shirley 2011; Shirley and Chiu 1997].

We were very surprised to find that simply changing the learning rate decay strategy can significantly improve the convergence of training, as shown in Fig. 1. We use cosine annealing warm restarts [Loshchilov and Hutter 2016] with an initial learning rate of 0.0005 and a minimum of 0.00005 for both neural textures and the network itself. During training, it decays the initial learning rate to the minimum and then restarts from the initial learning rate. We set the first restart at epoch 20, and the restart interval will be increased by a factor of two after each restart (i.e., the second restart at epoch 60 $(20 + 2 * 20)$, and the third at epoch 140 $(20 + 2 * 20 + 4 * 20)$).

We trained the models in the paper for at most 300 epochs. The whole training for 300 epochs takes about 18 hours on a single RTX 4090 GPU. However, the training time can be flexible. It can either be stopped early or continue based on the training loss, since different materials have different training complexities.

Table 1. This table summarizes the size and storage of our model. The positional feature plane has two settings: 400×400 for the measured BTF [Weinmann et al. 2014] and 800×800 for our synthetic materials.

	Height	Width	Channels	Storage
U Plane	400/800	400/800	8	4.88/19.53 MB
H Plane	50	50	8	78.12 KB
D Plane	50	50	8	78.12 KB
Network parameters				5.68 KB

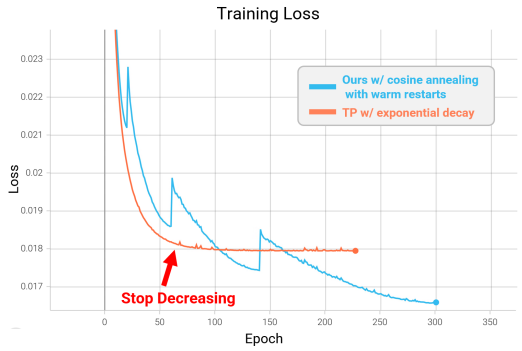

Fig. 1. The training comparison between ours and TP [Xu et al. 2024]. TP uses an exponential learning rate decay strategy, in which the loss stops decreasing at a high level, while we use a cosine annealing with warm restarts [Loshchilov and Hutter 2016] strategy as shown on the right, that gives a better convergency ability.

1.4 Inference & Rendering

Our rendering application is implemented in Nvidia's Falcor [Benty et al. 2020] real-time rendering framework. Since low-bit arithmetic is barely supported in common shader languages, we implement our Int8 inference in a custom CUDA kernel. Since our task is a large batch inference running at **thread level**, none of the efficient official CUDA GEMM libraries (e.g., CUTLASS) is suitable for us. They either lack thread-level APIs or do not support Int8 calculations.

[1] Some reviewers have observed a texture-sliding-like issue from the relighting example of leather11 BTF in our supplementary video. We suppose that is because we use the non-parallax-corrected version of leather11 data from UBO2014.

Author's address:

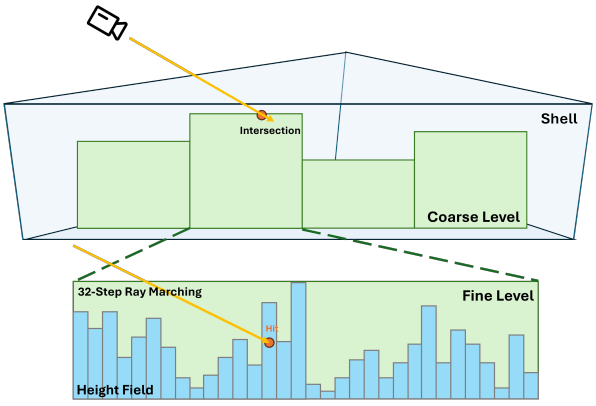

Fig. 2. Demonstration of our HF tracing process. Our HF tracing is based on a shell map. We first trace the ray at the coarse level inside the shell. Once we find an intersection, we step into the fine level and trace the ray started from the beginning of the coarse level's grid. We march at most 32 steps in the fine level and the hit point in the fine level will be the final hit position.

Therefore, we need to use native CUDA APIs only without relying on any other libraries.

Before running, we store the Int8 network weights by packing every four weights into a single Int32 value, creating an array of Int32 GPU buffers. Each texture plane is stored as a 2-layer float4 CUDA texture. The quantization scales are stored as constant variables.

In runtime, we first load the network weights into shared memory for each thread block to reduce high-latency global memory access. Then, we fetched features from the three textures based on the 6D input. These inputs are also packed into FP16 to reduce memory bandwidth. The fetched features are then quantized to Int8 and packed into Int32 for every four elements. We utilize CUDA's 8-bit integer 4-element vector dot product (dp4a) instruction for matrix multiplication. The outputs of each layer are in Int32 format, and they will be dequantized to FP16 and then re-quantized to Int8 for subsequent computations. For more specific details, please refer to our public code.

1.5 ACF Bézier Curve

To control the ACF, we use a cubic Bézier curve with four control points. The start and end points of the curve are set to $(0, 0)$ and $(1, 1)$ respectively, and only the two middle control points P_0 and P_1 are movable. Thus our cubic Bézier curve is defined as $C(t) = 3P_0t(1-t)^2 + 3P_1t^2(1-t) + t^3$, where t is from 0 to 1. By editing the two control points, we change the mapping from the initial ACF to the curved ACF. Given a normalized output of the initial ACF, we find the corresponding point on the Bézier curve by Newton's method, and thus get the curved ACF value.

1.6 Displacement Tracing

We design a two-step HF tracing based on a shell map. The tracing process is shown in Fig. 2. We only traverse between two LoD: coarse level and fine level. The coarse level served as an acceleration hierarchy to skip large empty spaces inside the shell. Once we find an intersection with the coarse level, we step into the fine level and trace the ray starting from the beginning of the coarse level's cell. We march at most 32 steps in the fine level and the hit point in the fine level will be the final hit position.

REFERENCES

Nir Benty, Kai-Hwa Yao, Petrik Clarberg, Lucy Chen, Simon Kallweit, Tim Foley, Matthew Oakes, Conor Lavelle, and Chris Wyman. 2020. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor https://github.com/NVIDIAGameWorks/Falcor.

Ilya Loshchilov and Frank Hutter. 2016. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983* (2016).

Alessandro Pappalardo. 2023. *Xilinx/brevitas.* https://doi.org/10.5281/zenodo.3333552

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 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Peter Shirley. 2011. Improved code for concentric map. https://psgraphics.blogspot.com/2011/01/improved-code-for-concentric-map.html

Peter Shirley and Kenneth Chiu. 1997. A Low Distortion Map between Disk and Square. *Journal of Graphics Tools* 2, 3 (1997), 45–52.

Michael Weinmann, Juergen Gall, and Reinhard Klein. 2014. Material Classification Based on Training Data Synthesized Using a BTF Database. In *Computer Vision – ECCV 2014*, David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer International Publishing, Cham, 156–171.

Zilin Xu, Zahra Montazeri, Beibei Wang, and Ling-Qi Yan. 2024. A Dynamic By-example BTF Synthesis Scheme. In *SIGGRAPH Asia 2024 Conference Papers (SA '24)*. Association for Computing Machinery, New York, NY, USA, Article 132, 10 pages. https://doi.org/10.1145/3680528.3687578