Instant Neural Representation for Interactive Volume Rendering

Qi Wu, David Bauer, Michael J. Doyle, and Kwan-Liu Ma

Abstract—Neural networks have shown great potential in compressing volume data for visualization. However, due to the high cost of training and inference, such volumetric neural representations have thus far only been applied to offline data processing and non-interactive rendering. In this paper, we demonstrate that by simultaneously leveraging modern GPU tensor cores, a native CUDA neural network framework, and a well-designed rendering algorithm with macro-cell acceleration, we can interactively ray trace volumetric neural representations (10-60fps). Our neural representations are also high-fidelity (PSNR > 30dB) and compact (10-1000× smaller). Additionally, we show that it is possible to fit the entire training step inside a rendering loop and skip the pre-training process completely. To support extreme-scale volume data, we also develop an efficient out-of-core training strategy, which allows our volumetric neural representation training to potentially scale up to terascale using only an NVIDIA RTX 3090 workstation.

Index Terms—Volume visualization, implicit neural representation, ray marching, path tracing.

1 INTRODUCTION

Modern simulations and experiments can produce massive amounts of high-fidelity volume data, which provide the necessary details for understanding complex scientific processes. These data are challenging to visualize interactively due to their sheer size. Neural networks have recently shown promising potentials for compactly and implicitly parameterizing continuous volumetric fields [1]. In the space of scientific visualization, this method has been applied to represent volume data by Lu et al. [2]. Their approach directly approximates the mapping from spatial coordinates to volume values using a multilayer perceptron (MLP). The trained MLP is then considered a compressed version of the original data. This representation is efficient because the memory footprint of a neural network is often orders of magnitude smaller than the original data. Sampling the representation is also flexible, as one can arbitrarily query volume values without explicit decompression and interpolation. However, complex MLPs (with hundreds of thousands of trainable parameters) are often needed to capture high-frequency details in the volume. The use of complex MLPs makes training and rendering prohibitively expensive. Furthermore, this method has not been tested on large-scale volume data generated by state-of-the-art simulations and experiments.

Recent progress in Neural Radiance Field (NeRF) has shown that smaller MLPs with high-dimensional trainable input encodings can accurately learn complex volumetric fields. This method allows us to capture high-frequency details in the data with relatively small models. In addition, by moving most of the network parameters to the encoding layer, training can be accelerated since each backward propagation process only updates a small number of parameters. Advancements in machine learning hardware and software can further accelerate training and inference.

In this work, we build on the successes of these techniques and introduce them to the scientific visualization domain. Our work models volume data using neural representations in the form of compact MLPs. It leverages the multi-resolution hash grid encoding method recently proposed by Müller et al. [3] to capture high-frequency details in the data. Then, we use the tensor-core-accelerated Tiny-CUDA-NN [4] machine learning framework and a pure CUDA implementation to maximize training speeds.

Although these training improvements are highly effective, significant gaps remain if we are to meet the typical performance and scalability requirements of volume visualization applications. Specifically, we must provide methods for the interactive rendering of these neural representations and methods for handling large-scale volume data.

To enable the interactive rendering of volumetric neural representations, we develop a sample streaming rendering technique. This sample streaming method iteratively interrupts the ray tracing kernel and streams out sample coordinates for network inference. We compare this method to a fully-customized in-shader inference routine, which is conceptually similar to the method used by Weiss et al. [5] and performs network inference inside shader programs. Our sample streaming method is about 3-8× faster than in-shader. Additionally, we introduce a macro-cell acceleration structure [6, 7] for rendering our neural representations. We demonstrate that this data structure is particularly suitable for accelerating this type of neural rendering and can bring up to 40× speedup. As a result, our sample streaming method can deliver highly interactive performance (10-60fps) and advanced global illumination via ray marching shadows and unbiased path tracing.

To provide a complete rendering solution for large-scale volume data, we develop out-of-core data streaming techniques. Training a neural network typically requires loading the entire dataset to GPU or system memory, which is infeas-
sible for large-scale data. A simple workaround would use virtual memory and file mapping. However, this approach is unsuitable for training volumetric neural representations, where data values are accessed randomly. For large-scale data, such a data access pattern can lead to enormous page swapping, leading to poor I/O performance. Our method maintains an ever-changing random subset of the data in memory and decouples data streaming from the network training. Thus, we can regularize the data access pattern. We demonstrate that this method is more than \( \times \) faster than the virtual memory implementation. We also demonstrate that it is feasible to fit a network training step inside our rendering loop and achieve online training—even for extreme-scale data.

An overview of our work is provided in Figure 1A. We also summarize our contributions as follows:

- Introduction of instant neural representation—a volume representation that can be 10-1000\( \times \) smaller than raw data and trained almost instantaneously using MLP and hash-grid encoding [3]—to the field of scientific visualization.
- A fast sample streaming algorithm to directly ray trace our neural representations interactively at 10-60fps, which significantly outperforms existing works.
- An efficient out-of-core sampling scheme that allows neural representation training to scale up to very large-scale on a single-GPU workstation for the first time.
- The demonstration and evaluation of interactive online training for volumetric neural representations.

2 RELATED WORK

Because we focus on creating compact neural representations for volume data, we first give an overview of related deep-learning methods for volume compression. Then, as our work uses multi-resolution hash grid encoding, we also provide the background of input encoding techniques.

Deep Learning for Volume Compression. High-quality volume visualization can be challenging, in part because handling large-scale, high-resolution volume data can be difficult. Therefore, deep learning techniques have been explored to compress volume data. Earlier work by Jain et al. [8] presented an encoder-decoder network to compress a high-resolution volume. Wurster et al. [9] later used a generative adversarial networks (GANs) hierarchy to complete the same task. Super-resolution neural networks can directly work with a low-resolution volume and upscale it when necessary. This technique can be helpful when it is too expensive to store the data in high resolution [10], [11], [12], or for all timesteps [13], [14], or both [14]. This technique can also accelerate simulations by allowing them to work on lower-resolution grids [15]. Lu et al. [2] explored using implicit neural representations, as mentioned previously. However, their method requires a time-consuming training process for every volume data. The concurrent work performed by Weiss et al. [5] improves this work by employing dense-grid encoding [16] for faster training and GPU tensor-core acceleration for faster inference. Doyub et al. [17] recently integrated implicit neural representation into the OpenVDB framework for handling high-resolution sparse volumes. Our work improves Lu et al.’s method but focuses on achieving interactive rendering algorithms and supporting large-scale volume data.

Network Input Encoding. We apply the idea of positional encoding that first maps input coordinates to a higher-dimensional space in our neural network, as it allows MLPs to capture high-frequency, local details better. One-hot encoding [18] and the kernel trick [19] are early examples of techniques that make complex data arrangements linearly separable. In deep learning, positional encodings are helpful for recurrent networks [20] and transformers [21]. In particular, they enclose vector positions as a sequence of sine and cosine functions. NeRF work and many others [22], [23], [24] use similar encoding methods, which are often referred to as frequency encodings. More recent works introduced parametric encodings via additional data structures such as dense grids [25], sparse grids [26], octrees [16], or multi-resolution hash tables [3]. By putting additional trainable parameters into the encoding layer, the neural network size can be reduced. Therefore, neural networks with such encoding methods can typically converge much faster while maintaining approximately the same accuracy. In this work, we adopt the multi-resolution hash grid method proposed by Müller et al. [3] because of its excellent performance in training MLPs.

3 NETWORK DESIGN

Volume data in scientific visualization essentially can be written as a function \( \Phi \) that maps a spatial location \((x, y, z)\) to a value vector \(v\). This data value at that spatial location \( \Phi : \mathbb{R}^3 \to \mathbb{R}^D, (x, y, z) \mapsto \Phi(x, y, z) = v \). Such a volumetric function is typically generated by simulations or measurements and then discretized, sampled, and stored. In this work, we focus on scalar field volume data \((D=1)\) and employ an implicit neural network to model the volumetric function \( \Phi \) by optimizing it directly using sample coordinates and values. Since the network is defined over the continuous domain \(\mathbb{R}^3\), it can directly calculate data values at an arbitrary spatial resolution and avoid explicit interpolations. Moreover, since the network processes each input position independently, data values can also be sampled on demand. Additionally, because the neural network approximates the volume data analytically, the number of
neurons needed to represent the volume data faithfully does not increase linearly as the data resolution increases, promising greater scalability potentially. Finally, although only scalar volumetric fields are studied in this work, we believe that the same method can be easily extended to multivariate cases. Weiss et al. [5] have reported findings in this direction by jointly training gradients and curvatures.

In practice, we represent $\Phi$ as an MLP following the approach used by Lu et al. [2]. However, unlike NeRF [27], which uses the network to predict RGB$\Phi$ values for rendering directly, we focus solely on the mapping from coordinate to volume values without transfer function classifications. We do this for two reasons. Firstly, such a neural representation can also be used for non-rendering tasks such as data analysis. Secondly, performing transfer function classification independently of the training process allows transfer functions to be applied freely and edited interactively.

### 3.1 Network Implementation

We implemented our neural representation in CUDA and C++ using the Tiny-CUDA-NN machine learning framework [4]. Unless specified otherwise, the “fully-fused-MLP” is used in this paper. By fitting inputs into the shared memory and weights into registers, this MLP implementation can more efficiently utilize GPU tensor cores. Additionally, as indicated by Mildenhall et al. [27] and Tancik et al. [22], an input encoder that first converts coordinates to high-dimensional feature vectors is needed. Such an encoder allows the network to capture high-frequency features in the data more efficiently. We find that the multi-resolution hash grid encoding method [3], initially proposed for NeRF and other computer graphics applications, is very effective for scientific volume data. Therefore we employ this method in our implementation.

As illustrated in Figure 1B, this hash grid encoding method defines $m$ logical 3D grids over the domain. Each grid is referred to as a level and is constructed using a hash table of size $T$. Each table entry stores $n$ trainable parameters (also referred to as features). Thus, each logical grid point can be mapped to a feature vector of size $n$ via the associated hash table. For a given input coordinate, a feature vector of size $n$ can be calculated for each level by interpolating nearby grid features. The final feature vector of size $N=n \times m$ can then be constructed via concatenation. We use this feature vector as the input for the MLP. This input encoding method is key to how our neural representation can support high-fidelity volume rendering.

Throughout this paper, we use ReLU activation functions for all MLP layers. We use the Adam optimizer [28] with an exponentially decaying learning rate to optimize networks. We use L1 loss unless specified otherwise. For each training step, we take precisely 65536 training samples.

### 4 Ablation Study

It is essential to understand how the choice of encoding method and hyperparameters, such as the number of encoding features or MLP layers, would influence the model’s performance. This understanding allows us to make an informed choice and choose the best combination of values.

#### 4.1 Input Encoding Method

We first investigated the effectiveness of the multi-resolution hash grid encoding. To do so, we compared the hash grid encoding with four widely used encoding strategies: dense grid, one-blob [24], frequency [27], and identity (i.e., no encoding). The dense grid encoding is similar to the hash grid encoding but uses a complete octree to store parameters. One-blob encoding generalizes the one-hot encoding by using a Gaussian kernel to activate multiple entries. The frequency encoding is also not trainable, and it encodes a scalar position as a sequence of sine and cosine functions. We provide the specific network configuration for each test in Figure 2. To make the comparison fair, we used around 900,000 trainable parameters in each network. To match the number of parameters used by grid-based encoding methods, we used much larger MLPs for other tests. Since such a large MLP cannot fit into registers, the fast fully-fused-MLP implementation does not work. So the slower CUTLASS-based MLP implementation was used instead as a compromise. This compromise also means that the training times reported in Figure 2 are naturally longer than other experiments performed in this work. We used the 1atmTemp data from a flame simulation at pressures of 1 atm [29] in this study. We also used the heat release field (HR) and the 10 atm variants in this paper.

The comparison between identity, frequency, and one-blob encoding indicates that mapping input coordinates to a higher-dimensional space allows the volume representation to extract more high-frequency details. This is in line with the effects observed in other domains [3]. By comparing grid-based encoding methods with the others, we found that using a grid-based encoding could not only improve training quality, but significantly reduce training time. Finally, the comparison between two grid-based encoding
methods showed that hash grid encoding could produce higher reconstruction quality and fewer stripe artifacts. This came at the cost of a slightly longer training time. However, hash grid was clearly the winner in terms of memory footprint. Summarizing all the findings, we believe the multi-resolution hash grid encoding is the best encoding strategy for constructing volumetric neural representations.

4.2 Hyperparameter Study

Next, we studied the effects of neural network parameters on network capacity. To perform this study, we used the 1atmHR data and hash grid encoding with a fixed hash table size $T = 2^{19}$. The reason to fix of $T$ is that the effect of $T$ has already been well-studied by Müller et al. [3]. We use the same GPU model in this paper, so we refer to their findings here and set $T = 2^{19}$. Then we scanned through other parameters. They are the number of encoding levels, the number of features per encoding level, the number of neurons per MLP layer, and the number of hidden layers in MLP. We report the reconstruction quality after training in terms of PSNR. For each configuration, the network was optimized for a sufficient amount of steps (at least 200k) until it practically converges. We believe the PSNR computed at this point reflects the network capacity. Figure 3 illustrates our results.

From the line plots shown in Figure 3 we can see that two encoding parameters (# of levels, # of features) produced a more significant impact on network capacity. This is understandable as most of the trainable parameters are stored in the encoding layer. However, both parameters also suffered from the law of diminishing returns. For the tested data, having 4-8 features per level and 4-8 encoding levels seemed to be the sweet spot. Beyond that, the extra benefits were limited. As for the MLP, it seemed that having 4-8 hidden layers was generally enough. Having more hidden layers did not bring any extra performance benefits and could even lead to performance reductions. Having more neurons in each MLP layer was generally beneficial, but the benefits were limited. However, this was likely because the MLPs used were all relatively small and thus only produce a limited impact.

Additionally, smartly picking network parameters could sometimes significantly improve training time and memory footprint for our volumetric neural representation. In Figure 3 we optimized two networks under identical conditions and compared their reconstruction qualities using PSNR of reconstructed volumes, perceptual qualities using LPIPS [30] of rendered images, training time, and memory footprint in terms of parameter count. We constructed the large network with the most encoding levels (=16), features per encoding level (=8), neurons per MLP layer (=64), and a high number of hidden layers (=8). The small network reduced the number of encoding levels and MLP hidden layers by 50% following our previous findings. Our small network achieved nearly the same quality. However, it required less than half of the training time and less than a quarter of the memory footprint.

5 TRAINING

We describe our training in terms of training steps. Note that a training step is different from an epoch. The latter indicates one complete pass of the training set through the algorithm. In our context, the training set is an infinite and continuous domain; thus, the former term is used. A single step is conducted by first generating a batch of random coordinates (batch size = 65536) within the normalized domain $[0, 1]^3$. Then we reconstruct the corresponding scalar value for each coordinate using the ground truth. The returned scalar value is normalized to $[0, 1]$. This paper refers to this step as the “sampling” step. Since all volumetric function values are generated independently, the sampling step is embarrassingly parallelizable. We use a normalized domain and range here because we observe that coordinates and values with high dynamic range can make the network very unstable, if not impossible, to optimize. Once the sampling is done, the input and the ground truth data are loaded into the GPU memory (if they are not already there) and then used to optimize our volumetric neural representation using the Tiny-CUDA-NN’s C++ API.
When the target volume is larger than the GPU memory, we can offload the sampling process to the CPU and reconstruct training samples with the help of multi-threading and SIMD vectorization. In our case, we implement our CPU-based sampling process using the OpenVKL [31] volume computation kernel library. As OpenVKL already supports a variety of volume structures, we can also apply our volumetric neural representations beyond dense grids. Our preliminary study in this direction also shows promising results. However, as this is mainly outside the scope of this paper, we leave the study and evaluation of sparse volumes to future works.

5.1 Out-of-Core Sampling

Modern scientific simulations and experiments often produce high-resolution datasets. These datasets often cannot be loaded into even the system memory. Thus, the sampling methods mentioned previously will not work. Using virtual memory and file mapping can be a workaround. However, as we draw training samples randomly, this workaround would lead to enormous page swappings and produce significant latencies. We develop a novel way to handle this situation, which we call out-of-core sampling. As illustrated by Figure 4, this method maintains a buffer of $R$ randomly selected 3D blocks of size $B$ in the system memory. Each 3D block in the buffer can be randomly mapped to a 3D region in the volume data domain via asynchronous data streaming. The first sampling step will properly initialize all the blocks to “warm up” the random buffer. Then we refresh only $S$ blocks (obviously $S \leq R$) in every subsequent sampling step.

To take samples, instead of uniformly sampling random coordinates, we first randomly select a block within the random buffer. Then randomly select a voxel within the selected block and calculate the corresponding voxel index. Finally, we jitter the voxel center by half a voxel and use the jittered $(x_c + X, y_c + Y, z_c + Z | X, Y, Z \sim U_{[-0.5,0.5]})$ coordinate as the sample. The corresponding sample value is then interpolated using values of the voxel and its neighbors. When trilinear interpolation is enabled, each block will be then interpolated using values of the voxel and its neighbors.

Although we implement the mapping via asynchronous I/O, we still perform a synchronization step before the start of each sampling process. This synchronization step allows us to simplify our sampling implementation to future works. Our implementation uses an NVMe SSD as the data storage device because it offers superior random read bandwidth. We default $S=1024$ to maintain a relatively good sampling performance and a reasonably large $R=65536$ for a good initial guess of the full-resolution data. We evaluated the effectiveness of out-of-core sampling and the choice of $S$ and $R$ in Section 7.2.5

5.2 Online Training and Rendering

With the help of Tiny-CUDA-NN, hash grid encoding, and our optimized sampling implementation, it is now possible to fit the entire training process inside the rendering loop while still achieving interactive visualization. Such an online-training capability allows users to interactively and visually examine the neural representation optimization process. It can also potentially accelerate the debugging and hyperparameter tuning process in practice. Online training also provides users with a new and elegant way to quickly preview extremely large datasets using a computer with limited RAM and VRAM. For example, the double-precision channel flow DNS [32] volume is around 950GB on disk. Traditionally, out-of-core streaming and progressive rendering techniques [33], [34] would be required to handle such data on a single machine. However, with online training, not only can the output of training be saved and reused in the future, but there is also no need to use a specialized progressive renderer. Since the training and rendering are entirely decoupled, the complexity of handling large-scale data is reduced.

6 Rendering

Decoding the volumetric neural representation back to a dense grid is a na"ive way to achieve interactive volume visualization. Such an approach would limit the ability to render large-scale volume data directly. The decoding process can be done progressively in every frame for online training. However, it can lead to noticeable rendering artifacts, especially in the early stage of training. Thus, we develop two novel techniques to overcome these drawbacks to render volumetric neural representations. For each technique, we provide the implementations of three rendering algorithms. They are ray-marching with the emission-absorption model, ray-marching with single-shot heuristic shadows [35] (only shading at the point of highest contribution), and volumetric path-tracing.

6.1 In-Shader Inference

The intuitive approach is to infer the neural network inside a shader program directly. Thus we develop a fully customized network inference routine in the Tiny-CUDA-NN framework using native CUDA. Specifically, our routine involves the following adjustments. First, we eliminate all the global GPU memory access and have the hash grid encoding layer fully fused into the MLP calculation. Inputs and outputs are managed locally within each threadblock and passed to the network via shared memory. Second, as illustrated by Figure 5, our routine breaks an MLP layer into 16-neuron-sized groups and allows the computation within each group to be completed by a 32-thread warp for 16 consecutive feature vectors. Thus, each warp only handles
6.2 Sample Streaming

The machine learning framework we use is optimized for batch training and inference, meaning that a relatively large number of inputs need to be pre-generated. Then the neural network infers all the inputs at the same time. This execution model can achieve higher GPU performance because it reduces control flow divergence, thus improving thread utilization. It can also effectively hide high-latency memory accesses as there are enough data for the processor to work on while waiting for a memory request. However, such an execution model can make the rendering algorithm more complex. To take full advantage of the batch inference execution model, we need to fundamentally change the rendering algorithm. Our implementations share the same spirit as wavefront path tracers [36]. However, we have to solve a very different challenge: volume densities are unknown in advance and expensive to compute.

In order to avoid a lengthy discussion of preliminaries, we assume basic knowledge of the structure of a modern volumetric ray marcher and path tracer. There are many implementations available publicly, such as OSPRay [37].

6.2.1 Ray Marching

We start by focusing on ray marching and splitting the algorithm into three smaller kernels: the ray-generation kernel, the coordinate-computation kernel, and the shading kernel (as highlighted in red in Figure 6). The ray-generation kernel initializes all the primary rays and intersects them with the volume. If no intersection is found, the ray is invalidated. We remove invalid rays via stream compaction (green in Figure 6). This operation is also performed for subsequent iterations in the render pass. Within the loop, the next \( K \) sample coordinates for each ray are streamed out for network inference. Then, inferred sample values are retrieved by another kernel for shading. Finally, the loop terminates if all the rays are invalid.

When the number of samples generated by each iteration \( K \) is greater than 1, the number of iterations needed to complete a frame can be reduced. This optimization reduces CUDA kernel launch overheads. Additionally, because exited rays are removed at the end of each iteration, the CUDA kernel size within each iteration will decrease as the loop iterates. Increasing \( K \), small tailing kernels can be batched up to reduce the launch overhead further. However, if \( K \) is too large, the number of samples being computed per iteration can be more than necessary because many rays might have exited earlier. These unused samples can only be discarded, and the time spent to compute these samples is
6.2.2 Path Tracing

The path tracing implementation is more complicated than ray marching. This is because a typical implementation would require sampling the volume at least twice per rendering pass: once for distance sampling (we use Woodcock tracking \[38\]) and one or more times for estimating the scattering (we use Woodcock tracking \[38\]). Then it switches to a scattering ray and continues propagation within the volume (again via Woodcock tracking) until the next valid interaction is found. When a valid interaction is not found, the ray either stays in the null-collision state or terminates if it exits the volume or is terminated by Russian Roulette \[39\].

Our implementation only maintains one active ray and iteratively changes the role of the ray among scattering, shadow, and null-collision. After finding a valid interaction between the photon and a particle, a shadow ray will be constructed to sample the light source (transmittance estimation \[39\] via Woodcock tracking \[38\]). Then it switches to a scattering ray and continues propagation within the volume (again via Woodcock tracking) until the next valid interaction is found. When a valid interaction is not found, the ray either stays in the null-collision state or terminates if it exits the volume or is terminated by Russian Roulette \[39\].

We implement our algorithm using two primary functions, as shown in Figure 7. The sample function computes the next sample coordinate for the current ray and potentially converts a shadow ray to a scattering ray. The shade function handles the rest. We have verified our path tracing implementation by comparing pixel-wise with a reference mega-kernel implementation.

6.3 Macro-Cell Optimization

Since the cost to evaluate a volume sample using a neural representation is high, we should take as few samples as possible to maximize the performance. To achieve that, we employ an acceleration structure called macro-cells \[6\]. \[7\]. A macro-cell contains \(\lceil D_1 \rceil \times \lceil D_2 \rceil \times \lceil D_3 \rceil\) voxels with \(D_1\) being volume dimensions. In our implementation, macro-cells are disjoint and closely packed to form a spatial partitioning grid. In each macro-cell, we store the value range of contained voxels and their maximum opacity \(\mu\) for the current transfer function.

We use a 3D digital differential analyzer (DDA) to traverse the ray through these macro-cells. For ray marching, we still perform regular sampling within each cell but adaptively vary the sampling step size \(s\) based on \(\mu\). We use the formula recently proposed by Morrical et al. \[40\] to calculate \(\bar{s} = \max\{s_2 + (s_2 - s_1) \cdot \min(\mu, 1 - 1/p, s_1)\}\), where we set the minimum step size \(s_1=1\), maximum step size \(s_2=64\), and \(p=2\). Sampled opacity \(\alpha\) is also corrected: \(\bar{\alpha} = 1 - (1-\alpha)^{s/s_1}\).

We can achieve a similar optimization for path tracing by constructing tight majorant bounds \[6\], \[7\]. We refer to the algorithm by Hofmann et al. \[41\] and start with a target optical thickness \(\tau_{\text{target}} = \ln(1-\epsilon), \epsilon \sim U[0,1]\). Then we visit all the macro-cells along the ray and accumulate their optical thicknesses \(\tau_i = \sum_{n=0}^{n_i} \mu_i \times s_i\), with \(s_i\) being the length of the \(i\)th ray segment. We stop when \(\tau_i > \tau_{\text{target}}\) for the first time and step backward to find the exact hit.

Typically, the value range data in macro-cells are precomputed before rendering by iterating over all voxels.
TABLE 1

| Dataset        | Single Shot Shadow | Path Tracing |
|----------------|--------------------|--------------|
|                | w/o MC | w/ MC | w/o MC | w/ MC |
| 1atmTemp       | 3.6 > 7.4 | 13 | 42 | 0.37 | 2.9 | 20 | 14 |
| 10atmTemp      | 3.4 > 7.8 | 12 | 37 | 0.37 | 2.9 | 2.5 | 17 |
| 1atmHR         | 6.0 > 9.5 | 20 | 55 | 0.58 | 2.9 | 3.2 | 14 |
| 10atmHR        | 4.8 > 7.4 | 14 | 43 | 0.53 | 2.9 | 2.0 | 8.2 |
| Chameleon      | 1.1 > 1.6 | 8.1 | 15 | 0.25 | 0.79 | 14 | 42 |
| MechHand       | 2.1 > 5.4 | 5.2 | 23 | 0.54 | 2.8 | 2.8 | 14 |
| SuperNova      | 2.6 > 3.8 | 19 | 29 | 0.81 | 1.9 | 11 | 32 |
| Instability    | 0.49 > 1.2 | 18 | 53 | 0.09 | 0.43 | 2.8 | 10 |
| PigHeart       | 0.70 > 1.8 | 22 | 57 | 0.16 | 0.77 | 4.9 | 20 |
| DNS (dp)       | 0.49 > 1.1 | 12 | 25 | 0.11 | 0.63 | 2.0 | 11 |

Such a pre-computation is also possible for online training. However, the purpose of online training would be slightly defeated, especially when we use online training as a tool to preview large-scale data. We propose a novel solution to this challenge. We observe that a neural representation is essentially a regression model constructed using all training samples. Thus, we can reuse the training samples generated in each training step to update the macro-cell value range field. Because the update process is embarrassingly parallelizable, the overhead is negligible. As we do not expect to infer a neural representation in an area where no training samples have been taken, online constructed value ranges should provide sufficient accuracy for rendering.

We evaluated our online macro-cell construction method by comparing rendered images with reference images and images rendered using pre-computed value ranges (Figure 8). The model and online constructed macro-cells were trained for precisely 10k steps. Results indicated that the images rendered with online constructed macro-cells were slightly less accurate (~0.1dB in PSNR). Online macro-cell construction is included in all experiments by default.

7 Evaluation

We evaluated our implementations on a Windows machine with an NVIDIA RTX 3090, an Intel 10500, 64GB RAM, and NVMe Gen3 SSDs. We used several datasets with various sources, contents, and sizes. Because Instability and PigHeart could not be loaded into GPU memory, we used the OpenVKL-based variant to generate samples. Our out-of-core sampling technique was used for the 0.95TB channel flow DNS dataset [32].

7.1 Rendering

In Table 1, we report the rendering performance of both rendering techniques we developed—in-shader and sample streaming. For each technique, three rendering modes were visited—ray marching with the emission-absorption lighting model (in the appendix), ray marching with single-shot heuristic shadows, and volumetric path tracing with next-event estimation. We recorded the performance with and without macro-cell optimization for each rendering mode. We also used the same rendering configuration, camera angle, and transfer function for each dataset for experiments in Figure 10 and Table 3.

From the results, we derived two main insights. Firstly, sample streaming methods were 3-8× faster than in-shader rendering methods because the sample streaming method could allow the network inference step to exploit more parallelism using extra GPU threadblocks. However, this is generally impossible for in-shader rendering, as one ray is managed by exactly one GPU thread. Secondly, macro-cells also brought significant speedups in performance. For in-shader methods, we observed 2-10× speedups. For sample streaming methods, we could get speedups as high as 40×. This is because macro-cells allow the renderer to traverse through empty and low-opacity regions quickly; thus, fewer volume samples are queried. For rendering neural representations, this is crucial as the cost to compute each volume sample is very high.

7.2 Training

We evaluated the training process by investigating the training time, model size, and volume reconstruction quality (PSNR and SSIM). A small hyperparameter scan was performed on each dataset to find a good neural representation. We scanned MLPs with 16, 32, and 64 neurons with 2-6 hidden layers. We also scanned the hash grid encoding with 4 and 8 features per level, the number of levels from 6-12, and the hash table size from 2^16 to 2^19 by incrementally changing the power term. We trained each configuration for 2000 steps (equivalent to 3-10s) and selected the best-performing configuration in terms of PSNR. In practice, performing such a parameter scan is not strictly necessary, as a guess based on our ablation study (Section 4) is often good enough. To verify this, we also trained all the datasets using a single configuration (detailed in the appendix) and recorded their performance and training time. We discuss the results in Section 7.2.1.

After identifying appropriate settings, we optimized two sets of neural representations on each dataset. The first set was optimized for 200k steps. All the models were extensively optimized at this point. Then, we created the second set of models with the same configuration but optimized them with 10× fewer training steps (20k-steps). This was to study how well the models could perform with a limited training budget. The training times of all the 20k-step models are reported in Table 2. We also report each volume’s model size and the calculated compression ratio.

Although our out-of-core sampling method enables us to train the terascale DNS dataset efficiently, it is still significantly slower than other in-core training methods. Using the brute force method to find the best configuration would be too costly. Therefore, we manually identified a good configuration for DNS. We also used L2 loss because we found it to provide more training stability in conjunction with an increased number of training parameters.

Fig. 9. The value distribution of PigHeart (left), the low-dynamic-range version (LDR) of PigHeart (left, shaded region), and 10atmHR (right).
TABLE 2
Training results and comparisons with Neucomp [2]. All the volumes are single-precision floating point volumes except the double-precision DNS dataset. TTHRESH is the state-of-the-art traditional volume compression algorithm. TTHRESH experiments targeted our 20k-Models’ PSNRs. Because the TTHRESH algorithm is CPU-based, these experiments were performed on a 88-core server with 256GB memory.

| Dataset       | Dimension | Model Size | Time (s) | Ratio | 20k-Model PSNR | MSSIM | 20k-Model PSNR | MSSIM | TTHRESH Time (s) | Ratio | PSNR | MSSIM |
|---------------|-----------|------------|----------|-------|----------------|-------|----------------|-------|-----------------|-------|------|-------|
| 1atmTemp      | 1152 x 320 x 853 | 44.6MB     | 39.6s    | 28:1  | 34.4           | 0.964 | 38.4           | 0.981 | 136.1          | 26.7s | 60.8 | 158.1 |
| 10atmTemp     | 1152 x 426 x 853 | 44.6MB     | 39.5s    | 32:4  | 32.6           | 0.965 | 36.4           | 0.982 | 176.1          | 29.4s | 57.9 | 259.4 |
| 1atmHR        | 1152 x 320 x 853 | 28.6MB     | 31.7s    | 28:1  | 31.9           | 0.976 | 36.8           | 0.982 | 133.1          | 12.9s | 31.7 | 28.2 |
| 10atmHR       | 1152 x 426 x 853 | 28.6MB     | 31.6s    | 32:4  | 31.9           | 0.976 | 36.8           | 0.982 | 133.1          | 12.9s | 31.7 | 28.2 |
| Chameleon     | 1024 x 1024 x 1080 | 28.6MB    | 24.8s    | 48:9  | 49.8           | 0.997 | 54.3           | 0.998 | 587.1          | 789.4s | 57.9 | 259.7 |
| MechHand      | 640 x 220 x 229  | 6.7MB      | 25.8s    | 39:1  | 39.1           | 0.997 | 41.5           | 0.998 | 34.1           | 12.9s | 34.1 | 12.9 |
| SuperNova     | 432 x 432 x 432  | 8.6MB      | 17.9s    | 49:9  | 49.9           | 0.997 | 52.2           | 0.998 | 79.1           | 64.6s | 79.1 | 64.6 |
| Instability   | 2048 x 2048 x 1920 | 52.6MB    | 96.5s    | 27:9  | 27.9           | 0.941 | 31.6           | 0.961 | out-of-memory   | out-of-memory |
| PigHeart      | 2048 x 2048 x 2612 | 6.6MB     | 78.5s    | 52.9  | 52.9           | 0.996 | 55.0           | 0.997 | out-of-memory   | out-of-memory |
| DNS           | 10240 x 7680 x 1536 | 1.5GB     | 6706.0s  | 34.6  | 34.6           | 0.917 | 35.9           | 0.922 | out-of-memory   | out-of-memory |

Fig. 10. Rendering quality comparison. Neural representations obtained from the previous experiment shown in Table 2 were used. We rendered each neural representation by exactly 200 frames. The image space PSNR, SSIM, and LPIPS [30] were computed against reference renderings.

7.2.1 Reconstruction Quality
We measured how well our volumetric neural representation could learn features from a given volume data by comparing the reconstructed volume (Table 2) and rendered images (Figure 10) with the ground truth. The appendix can find rendering results for datasets not shown in Figure 10.

We progressively decoded the volume at its original resolution and calculated PSNR and the mean structural similarity index (MSSIM) between the reconstructed volume and the original data. Results for both 20k and 200k models are reported in Table 2. We also rendered both models using our sample streaming path tracer for 200 frames and compared image differences in Figure 10 in terms of PSNR, MSSIM, and LPIPS. Our neural representations (200k step models) could provide good volume reconstruction and rendering accuracy. With a limited training budget (20k step models), our neural representations could still perform reasonably well, especially for tasks like rendering, where decreases in image accuracy were tiny.

As mentioned previously, we also trained each dataset with a fixed network configuration. Results are shown in Table 3. We could still get nearly identical reconstruction quality without a brute-force parameter scan. The training times were slightly different compared to previous results. This is expected as the number of parameters changed significantly for some data.

7.2.2 Compression Ratio
In Table 2, we highlight the memory footprints of our neural representations and the corresponding compression ratios. The DNS data is double precision, whereas others are single-precision floating-point volumes. Overall, our method could faithfully compress data by 10-1000×, especially for large datasets such as Chameleon (4.5GB), Instability (32GB), PigHeart (44GB), and DNS (950GB).

Notably, the neural representation optimized for the PigHeart data presented a surprisingly high compression ratio (6640:1). Our investigation suggests that the PigHeart data has a fairly uneven value distribution (left image in Figure 9). This will make the reconstruction error relatively minor compared with more evenly distributed data such as 10atmHR (right image in Figure 9). To verify our claim, we clamped the dynamic range of PigHeart volume to [0, 2000] (referred to as PigHeart (LDR), with its data distribution
CPU-based sampling, the sampling process was slower, and based sampling and out-of-core sampling methods. For slower to train. This is because they are trained with CPU-based sampling, the sampling process was slower, and the overhead of constantly copying training samples from CPU to GPU was not negligible. For out-of-core sampling, the performance was primarily bounded by the disk I/O latency. We further investigated this and compared the out-of-core sampling method with the method directly using virtual memory via the CreateFileMapping API (with uniformly sampled random coordinates). We used both methods with trilinear interpolation to train the same model for 10k steps. The virtual memory method took 12.7 hours to complete. Our out-of-core sampling method only took 54 mins (∼14× faster). This is because randomly and uniformly accessing coordinates would lead to a considerable number of page faults, which in turn cause I/O overhead.

In Table 3 we also demonstrate the performance of online training. As we can see, for all the datasets, interactive performance was achieved. The breakdown of training and rendering latency indicates that, with our settings, all the online training processes were rendering-bound, with each training step only making up for a fraction of the total time. The only exception was DNS, which used the out-of-core sampling method for training. However, this is understandable as this technique requires a lot of high-lateney I/O operations.

In Figure 11, we compared a neural representation (optimized for 10k steps, configuration in the appendix) with a naively downsampled version of the same dataset. The downsampling factor was tuned to achieve a similar level of compression. We compared the rendering of both versions to the ground truth. Renderings were done using our sample streaming path tracer. The image rendered from our neural representation could generally match the ground truth result, with minor artifacts visible only after zooming in. In contrast, the downsampled volume failed to capture many details of the data and produced a noticeable color shift. This was because high- and low-value features could be averaged out during downsampling.

7.2.3 Comparison with TTHRESH and Neurcomp
Additionally, we compared our method with Neurcomp and the state-of-the-art traditional volumetric compression technique TTHRESH. For TTHRESH, each dataset was compressed to the same PSNR as our 20k-step model. As TTHRESH is a CPU-based algorithm, we performed all the compressions on a dual Intel Xeon E5-2699 (88-core) server with 256GB main memory. We report our results in Table 2. We found that TTHRESH performed much better in terms of compression ratio. However, it took the algorithm significantly longer to compress most datasets, and the compressed volume cannot be accessed or rendered without an explicit decompression. For Neurcomp, eight 256-neuron residual blocks were used for all the datasets. We could not use larger networks because, firstly, larger Neurcomp networks (e.g., 300-1000 neurons per block) were extremely difficult to optimize. Secondly, the 256-neuron network could already deliver comparable reconstruction quality compared to our fix-sized network (Table 3). Moreover, Neurcomp networks achieved better compression ratios but took hours to train. This performance difference is significant even after accounting for the difference between Python and CUDA.

7.2.4 Training Time
We compared the training of 20k models shown in Table 2. In general, we observed two trends. Firstly, smaller models were generally faster to train. This is because there are fewer gradients to compute in each training step. Secondly, Instability, PigHeart, and DNS data were significantly slower to train. This is because they are trained with CPU-based sampling and out-of-core sampling methods. For CPU-based sampling, the sampling process was slower, and the overhead of constantly copying training samples from CPU to GPU was not negligible. For out-of-core sampling, the performance was primarily bounded by the disk I/O latency. We further investigated this and compared the out-of-core sampling method with the method directly using virtual memory via the CreateFileMapping API (with uniformly sampled random coordinates). We used both methods with trilinear interpolation to train the same model for 10k steps. The virtual memory method took 12.7 hours to complete. Our out-of-core sampling method only took 54 mins (∼14× faster). This is because randomly and uniformly accessing coordinates would lead to a considerable number of page faults, which in turn cause I/O overhead.

In Table 3 we also demonstrate the performance of online training. As we can see, for all the datasets, interactive performance was achieved. The breakdown of training and rendering latency indicates that, with our settings, all the online training processes were rendering-bound, with each training step only making up for a fraction of the total time. The only exception was DNS, which used the out-of-core sampling method for training. However, this is understandable as this technique requires a lot of high-latency I/O operations.

In Figure 11, we compared a neural representation (optimized for 10k steps, configuration in the appendix) with a naively downsampled version of the same dataset. The downsampling factor was tuned to achieve a similar level of compression. We compared the rendering of both versions to the ground truth. Renderings were done using our sample streaming path tracer. The image rendered from our neural representation could generally match the ground truth result, with minor artifacts visible only after zooming in. In contrast, the downsampled volume failed to capture many details of the data and produced a noticeable color shift. This was because high- and low-value features could be averaged out during downsampling.

7.2.5 Sampling Pattern
Different sampling strategies might lead to differences in training quality. To study this relationship, we conducted three experiments. In the first one, we trained the 10atmTemp data with the exact neural representation using in-core, out-of-core (default settings as mentioned in Section 5.1), and virtual memory sampling methods. For each sampling method, we tested both trilinear and nearest neighbor interpolations. The loss curve of each training could be found in Figure 12A-C. We observed that experiments using the same interpolation method all generally matched each other. However, trilinear interpolation allowed training to perform significantly better than their nearest neighbor interpolation counterparts. Next, we investigated the large DNS dataset. Due to its size, we performed only out-of-core (default settings) and virtual memory sampling methods. Results are shown in Figure 12D. Although the out-of-core sampling method did not generate sample coordinates uniformly and randomly, it did not obviously impact the training. Finally, we trained the Instability with different out-of-core training settings (M and R). We found that larger values for M and R both positively impacted training quality.
networks. Therefore, applying such techniques to typical scientific visualization workflows is promising.

With this work, we have taken an essential step towards enabling real-time, memory-efficient volume visualization for terascale applications. We hope this project will spawn subsequent research to drive the exascale evolution of data processing.

**ACKNOWLEDGMENTS**

This project is sponsored in part by the U.S. Department of Energy through grant DE-SC0019486. The ammonia/hydrogen/nitrogen-air flames dataset is kindly provided by Dr. Martin Rieth. The channel flow simulation dataset is kindly provided by Dr. Myoungkyu Lee.

**REFERENCES**

[1] V. Sitzmann, J. Martel, A. Bergman, D. Lindell, and G. Wetzstein, “Implicit neural representations with periodic activation functions,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 7462–7473, 2020.

[2] Y. Lu, K. Jiang, J. A. Levine, and M. Berger, “Compressive neural representations of volumetric scalar fields,” in *Computer Graphics Forum*, vol. 40. Wiley Online Library, 2021, pp. 139–146.

[3] T. Müller, A. Evans, C. Schiel, and A. Keller, “Instant neural graphics primitives with a multiresolution hash encoding,” *arXiv preprint arXiv:2201.05989*, 2022.

[4] T. Müller, “Tiny CUDA neural network framework,” 2021 (Online), https://github.com/nvlabs/tiny-cuda-nn.

[5] S. Weiss, P. Hermüller, and R. Westermann, “Fast neural representations for direct volume rendering,” *arXiv preprint arXiv:2112.01579*, 2021.

[6] J. Novák, A. Selle, and W. Jarosz, “Residual ratio tracking for estimating attenuation in participating media.” *ACM Trans. Graph.*, vol. 33, no. 6, pp. 179–1, 2014.

[7] L. Szirmay-Kalos, B. Tóth, and M. Magdics, “Free path sampling in high resolution inhomogeneous participating media,” in *Computer Graphics Forum*, vol. 30, no. 1. Wiley Online Library, 2011, pp. 85–97.

[8] S. Jain, W. Griffin, A. Godil, J. W. Bullard, J. Terrill, and A. Varshney, “Compressed volume rendering using deep learning,” in *Proceedings of the Large Scale Data Analysis and Visualization (LDAV) Symposium*. Phoenix, AZ, 2017.

[9] S. W. Warster, H.-W. Shen, H. Guo, T. Peterka, M. Raj, and J. Xu, “Deep hierarchical super-resolution for scientific data reduction and visualization,” *arXiv preprint arXiv:2107.00462*, 2021.

[10] Z. Zhou, Y. Hou, Q. Wang, G. Chen, J. Lu, Y. Tao, and H. Lin, “Volume upsampling with convolutional neural networks,” in *Proceedings of the Computer Graphics International Conference*, 2017, pp. 1–6.

[11] J. Han and C. Wang, “SSR-TVD: Spatial super-resolution for time-varying data analysis and visualization,” *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[12] L. Guo, S. Ye, J. Han, H. Zheng, H. Gao, D. Z. Chen, J.-X. Wang, and C. Wang, “SSR-VFD: Spatial super-resolution for vector field data analysis and visualization,” in *Proceedings of IEEE Pacific Visualization Symposium*, 2020.

[13] J. Han and C. Wang, “TSR-TVD: Temporal super-resolution for time-varying data analysis and visualization,” *IEEE transactions on visualization and computer graphics*, vol. 26, no. 1, pp. 205–215, 2019.

[14] J. Han, H. Zheng, D. Z. Chen, and C. Wang, “STNet: An end-to-end generative framework for synthesizing spatiotemporal super-resolution volumes,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 1, pp. 270–280, 2021.

[15] Y. Xie, E. Franz, M. Chu, and N. Thueray, “TempoGAN: A temporally coherent, volumetric gan for super-resolution fluid flow,” *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, pp. 1–15, 2018.

[16] T. Takikawa, J. Litalien, K. Yin, K. Kreis, C. Loop, D. Nowrouzezahrai, A. Jacobson, M. McGuire, and S. Fidler, “Neural geometric level of detail: Real-time rendering with implicit 3d shapes,” in *Proceedings of the IEEE/ICCV Conference on Computer Vision and Pattern Recognition*, 2021, pp. 11 358–11 367.
[17] D. Kim, M. Lee, and K. Museth, “Neuralvdb: High-resolution sparse volume representation using hierarchical neural networks,” arXiv preprint arXiv:2208.04448, 2022.

[18] D. Harris and S. L. Harris, Digital design and computer architecture. Morgan Kaufmann, 2010.

[19] S. Theodoridis and K. Koutroumbas, Pattern recognition. Elsevier, 2006.

[20] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional sequence to sequence learning,” in International Conference on Machine Learning. PMLR, 2017, pp. 1243–1252.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” Advances in neural information processing systems, vol. 30, 2017.

[22] M. Tancik, P. Srinivasan, B. Mildenhall, S. Fridovich-Keil, N. Raghavan, U. Singhal, R. Ramamoorthi, J. Barron, and R. Ng, “Fourier features let networks learn high frequency functions in low dimensional domains,” Advances in Neural Information Processing Systems, vol. 33, pp. 7537–7547, 2020.

[23] J. T. Barron, B. Mildenhall, M. Tancik, P. Hedman, R. Martin-Brualla, and P. P. Srinivasan, “Mip-NeRF: A multiscale representation for anti-aliasing neural radiance fields,” in Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 5855–5864.

[24] T. Müller, B. McWilliams, F. Rousseau, M. Gross, and J. Novák, “Neural importance sampling,” ACM Transactions on Graphics (TOG), vol. 38, no. 5, pp. 1–19, 2019.

[25] J. N. Martel, D. B. Lindell, C. Z. Lin, E. R. Chan, M. Monteiro, and G. Wetzstein, “Acomo: Adaptive coordinate networks for neural scene representation,” arXiv preprint arXiv:2105.02788, 2021.

[26] S. Hadadan, S. Chen, and M. Zwickler, “Neural radiosity,” ACM Transactions on Graphics (TOG), vol. 40, no. 6, pp. 1–11, 2021.

[27] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, “Nerf: Representing scenes as neural radiance fields for view synthesis,” in European conference on computer vision. Springer, 2020, pp. 405–421.

[28] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” arXiv preprint arXiv:1412.6980, 2014.

[29] M. Rieth, A. Gruber, F. A. Williams, and J. H. Chen, “Enhanced burning rates in hydrogen-enriched turbulent premixed flames by diffusion of molecular and atomic hydrogen,” Combustion and Flame, p. 111740, 2021.

[30] R. Zhang, P. Isola, A. A. Efros, E. Shechtman, and O. Wang, “The unreasonable effectiveness of deep features as a perceptual metric,” in CVPR, 2018.

[31] Intel Corporation, “Intel open volume kernel library,” 2022 (Online), https://www.openvkl.org, version 0.9.0, accessed on 2022-03-09.

[32] M. Lee and R. D. Moser, “Direct numerical simulation of turbulent channel flow up to Reτ ≈ 5200,” Journal of Fluid Mechanics, vol. 774, pp. 395–415, Jul. 2015.

[33] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister, “Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach,” IEEE Transactions on Visualization and Computer Graphics, vol. 18, no. 12, pp. 2285–2294, 2012.

[34] Q. Wu, M. J. Doyle, and K.-L. Ma, “A Flexible Data Streaming Design for Interactive Visualization of Large-Scale Volume Data,” in Eurographics Symposium on Parallel Graphics and Visualization, R. Bujack, J. Tierny, and F. Sadlo, Eds. The Eurographics Association, 2022.

[35] C. Brownlee and D. DeMarle, Fast Volumetric Gradient Shading Approximations for Scientific Ray Tracing. Berkeley, CA: Apress, 2021, pp. 725–733. [Online]. Available: https://doi.org/10.1007/978-1-4842-7185-8_45

[36] S. Laine, T. Karras, and T. Aila, “Megakernels considered harmful: Wavefront path tracing on gpus,” in Proceedings of the 5th High-Performance Graphics Conference, 2013, pp. 137–143.

[37] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günter, and P. Navrátil, “Ospray—a cpu ray tracing framework for scientific visualization,” IEEE transactions on visualization and computer graphics, vol. 23, no. 1, pp. 931–940, 2016.

[38] E. Woodock, T. Murphy, H. P., and L. T.C., “Techniques used in the ospray code for Monte Carlo neutronics calculation in reactors and other systems of complex geometry,” Argonne National Laboratory, Tech. Rep., 1965.

[39] M. Pharr, W. Jakob, and G. Humphreys, Physically based rendering: From theory to implementation. Morgan Kaufmann, 2016.

[40] N. Morrical, W. Usher, I. Wald, and V. Pascucci, “Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing,” in 2019 IEEE Visualization Conference (VIS). IEEE, 2019, pp. 256–260.

[41] N. Hofmann and A. Evans, “Efficient unbiased volume path tracing on the gpu,” in Ray Tracing Gems II. Springer, 2021, pp. 699–711.

[42] P. Klacansky, “Open scientific visualization datasets,” https://klacansky.com/open-scivis-datasets/.

[43] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, “Tthresh: Tensor compression for multidimensional visual data,” IEEE transactions on visualization and computer graphics, vol. 26, no. 9, pp. 2891–2903, 2019.

[44] J. J. Park, P. Florence, J. Straub, R. Newcombe, and S. Lovegrove, “Deepsdf: Learning continuous signed distance functions for shape representation,” in Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 165–174.
APPENDIX A
NETWORK CONFIGURATIONS
In this section, we list all the network configurations used by experiments discussed in this paper. The specific documentation about each parameter can be found in the documentation page of Tiny-CUDA-NN [4].

A.1 Default Optimizer and Loss Function
Default optimizer and loss settings used in this paper, unless specified otherwise.

```json
"optimizer": {  
  "otype": "ExponentialDecay",  
  "decay_start": 2000,  
  "decay_interval": 1000,  
  "learning_rate": 0.005,  
  "n_hidden_layers": 4  
},  

"network": {  
  // CutlassMLP is slower, but can support more neurons  
  "otype": "CutlassMLP",  
  "n_neurons": 256,  
  "n_hidden_layers": 14  
},  

"encoding": {  
  "otype": "Identity",  
  "n_features_per_level": 8,  
  "n_levels": 8,  
  "n_hidden_layers": 3  
},  

"loss": {  
  "otype": "L1"  
}
```

A.2 Ablation Study: Encoding Comparison
The network configurations used in Section 4 and Figure 2.

```json
"network": {  
  "otype": "HashGrid",  
  "n_levels": 8,  
  "n_features_per_level": 4,  
  "n_hidden_layers": 4  
},  

"network": {  
  "otype": "DenseGrid",  
  "n_levels": 2,  
  "n_features_per_level": 2,  
  "n_hidden_layers": 37  
},  

"network": {  
  "otype": "Frequency",  
  "n_frequencies": 32  
},  

"network": {  
  "otype": "OneBlob",  
  "n_bins": 64  
}
```

A.3 Hyperparameter Study
Two highlighted networks in the hyperparameter study and Figure 3. Hash grid encoding and fully-fused MLP were used for both networks. A simplified description scheme is used for conciseness.

```json
// large network  
"params": {  
  "n_levels": 16,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 19,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 8  
}

// small network  
"params": {  
  "n_levels": 8,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 19,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 4  
}
```

A.4 Networks used for Training Benchmark (Table 2)

```json
// latmTemp  
"params": {  
  "n_levels": 10,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 19,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 3  
}

// 10atmTemp  
"params": {  
  "n_levels": 10,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 19,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 4  
}

// latmHR  
"params": {  
  "n_levels": 8,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 19,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 4  
}

// 10atmHR  
"params": {  
  "n_levels": 8,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 19,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 4  
}

// Chameleon  
"params": {  
  "n_levels": 8,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 19,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 4  
}

// MechHand  
"params": {  
  "n_levels": 11,  
  "n_features_per_level": 8,  
  "log2_hashmap_size": 17,  
  "base_resolution": 4,  
  "n_neurons": 64,  
  "n_hidden_layers": 5  
}
```
A.5 Benchmark with Fix-sized Networks (Table 3)

A.6 Compare with Down-Sampling (Figure 11)

A.7 Sampling Pattern Study (Figure 12)

A.8 Online Macro-Cell Construction (Figure 8)

APPENDIX B
ADAPTIVITY WITH TIME-VARYING DATA

Our neural representation method can also be applied to time-varying data directly with the help of our online-training capability. In Figure 13 we demonstrate the real-time loss curve when interactively visualizing the multi-timestep vortices dataset provided by Deborah Silver at Rutgers University. Because we want continuously train the same neural network (without resetting network weights) despite having a changeable target data, we did not train the network with learning rate decay in our experiment. Instead, we used a simple Adam optimizer with learning rate being fixed to 0.01. We changed the data timestep every 5 seconds. We can see that our neural network could quickly adapt to the new timestep, and automatically adjust what has been learned within around two seconds. Network used for Figure 13 is listed below.
Fig. 13. We can use a single volumetric neural representation to interactively visualize a time-varying dataset. In the experiment, the timestep was automatically incremented every 5 seconds.

APPENDIX C
COMPARE OUR SAMPLE-STREAMING ALGORITHM WITH THE RELATED WORK
Notably, Müller et al. [3] recently also proposed a volumetric path tracer involving implicit neural representations. We want to clarify the difference between their method and our sample streaming implementation. In their method, a radiance and density field is directly fitted using the noisy output of a volumetric path tracer. Then, a standard NeRF ray marcher is used to render the fitted neural representation. Because the in-scattering terms are learned already, the ray-marched final image can match the result of an unbiased volumetric path tracer. However, in their algorithm, the ground truth volume density field is always accessible; thus, the path tracer can scatter rays freely without leaving the render kernel. Later, their ray marcher also uses the ground truth density field for empty space skipping, as the neural representation has not been trained on these empty regions. In our case, the ground truth volume density is not used. Therefore, we base all of our calculations on the neural representation. This includes empty space skipping, shadows, multi-scattering, and direct lighting, which makes our implementation independent from the ground truth and significantly more compact (and more sophisticated).

APPENDIX D
TUNING OUR SAMPLE-STREAMING ALGORITHM
As mentioned in Section 6.2.1 for the sample-streaming ray marching algorithm, the rendering performance can be improved by allowing a ray to take more than one sample in each iteration ($K>1$). In Figure 14 we show the experiment to tune $K$. We can see that the algorithm performs best when $K$ was around 8, which is the value we used in this paper.

APPENDIX E
ADDITIONAL RESULTS AND IMAGES
Here we provide additional rendering results that are not shown in the paper. Table 4 illustrates the same experiment as Table 1 but with the simple ray marching algorithm included. Next, rendering quality comparisons of datasets that are not shown in Figure 10 can be found in Figure 16, Figure 17, and Figure 18. Finally, a larger version of Figure 12 can be found in Figure 15.
Table 4
Framerates in FPS of different rendering methods and datasets. Neural representations are from the Table 2 experiment.

| Dataset   | Ray Marching w/o MC | Ray Marching w/ MC | Single Shot Shadow w/o MC | Single Shot Shadow w/ MC | Path Tracing w/o MC | Path Tracing w/ MC |
|-----------|---------------------|-------------------|---------------------------|--------------------------|-------------------|------------------|
|           | IS  | SS   | IS  | SS   | IS  | SS   | IS  | SS   | IS  | SS   | IS  | SS   |
| 1atmTemp  | 4.6  | 13   | 20  | 69   | 3.6  | 7.4  | 13  | 42   | 0.37 | 2.9  | 2.0  | 14   |
| 10atmTemp | 4.7  | 14   | 18  | 60   | 3.4  | 7.8  | 12  | 37   | 0.37 | 2.9  | 2.5  | 17   |
| 1atmHR   | 8.4  | 17   | 30  | 89   | 6.0  | 9.5  | 20  | 55   | 0.58 | 2.9  | 3.2  | 14   |
| 10atmHR  | 6.5  | 12   | 24  | 67   | 4.8  | 7.4  | 14  | 43   | 0.53 | 2.9  | 2.0  | 8.2  |
| Chameleon | 1.2  | 2.0  | 10  | 18   | 1.1  | 1.6  | 8.1 | 15   | 0.25 | 0.79 | 14   | 42   |
| MechHand | 2.7  | 6.9  | 6.6 | 89   | 2.1  | 5.4  | 5.2 | 23   | 0.54 | 2.8  | 2.8  | 14   |
| SuperNova | 2.8  | 4.4  | 24  | 35   | 2.6  | 3.8  | 19  | 29   | 0.81 | 1.9  | 11   | 32   |
| Instability | 0.57 | 1.7  | 24  | 79   | 0.49 | 1.2  | 18  | 53   | 0.09 | 0.43 | 2.8  | 10   |
| PigHeart | 0.80 | 2.4  | 30  | 82   | 0.70 | 1.8  | 22  | 57   | 0.16 | 0.77 | 4.9  | 20   |
| DNS (dp) | 0.58 | 1.4  | 17  | 39   | 0.49 | 1.1  | 12  | 25   | 0.11 | 0.63 | 2.0  | 11   |
Fig. 16. A larger version of Figure 10.
Fig. 17. Additional rendering quality comparisons for datasets that are absent from Figure 10.
Fig. 18. Additional rendering quality comparisons for datasets that are absent from Figure 10.