Auto Rigging Through LBS-based Neural Rendering

ETHAN WEBER, ERICH LIANG, NIANXU WANG, and RUILONG LI, UC Berkeley

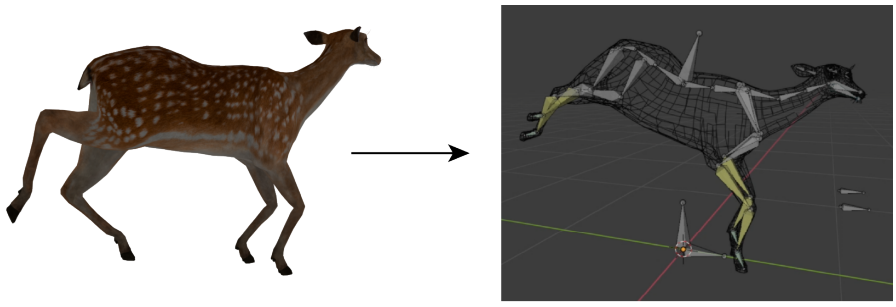

Fig. 1. Our method takes in multi-view images, and create a animatable rig of the avatar with optimized skeleton movements.

Rigging is the process of creating a digital avatar with skeletons to control its animation. A rig for an avatar includes its geometry, texture, a set of bones and the linear blend skinning (LBS) weights which describes how the geometry changes with respect to its bone movements. To create a vivid character, traditional methods usually involves a lot of manual efforts from artists to setup the aforementioned properties. Recent neural rendering techniques TAVA[1] make it possible to create a realistic digital avatar from multi-view images in the real-life given the accurate skeleton movements. Although some off-the-shell methods can extract human skeleton movements from images, they would never be super accurate. In this project, we aim at study the problem of reconstruct and optimize the skeleton movements along with all the other aforementioned properties from multi-view images to automatically create a digital avatar rig ready for animation. Some preliminary experiments in this milestone report demonstrate that given a relatively good initialization of the skeleton movements, we are able to optimize them towards better accuracy through neural rendering.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: neural rendering, pose optimization, linear blend skinning

ACM Reference Format:

Ethan Weber, Erich Liang, Nianxu Wang, and Ruilong Li. 2022. Auto Rigging Through LBS-based Neural Rendering. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2022, Woodstock, NY*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

[1]TAVA is Ruilong's recent submission "TAVA: Template-free Animatable Volumetric Actors"

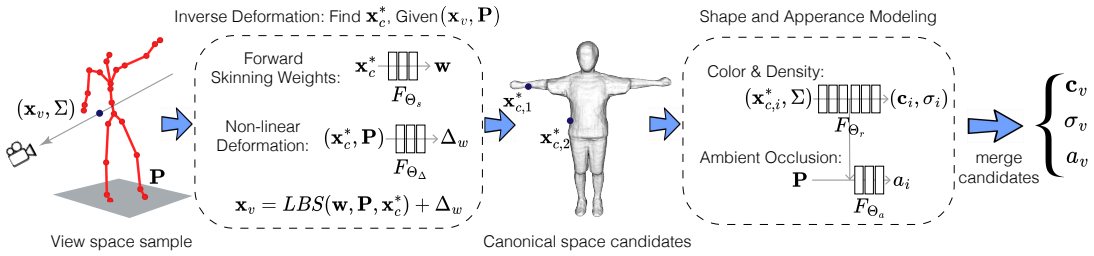

Fig. 2. *TAVA Overview*. TAVA uses volumetric rendering techniques to create the actor representation. For each sampled point, TAVA uses LBS based non-linear deformation combined with a blending weight model for which it identifies the root in the canonical space. In this space, TAVA use a color, density, and ambient occlusion model to parameterize the appearance.

1 INTRODUCTION

This project will be built on top of Ruilong's recent submission "TAVA: Template-free Animatable Volumetric Actors," which extends MipNeRF [1] to animatable objects using Neural Radiance Fields (NeRF) [2]. The pipeline of TAVA is shown in Figure 2. TAVA already succeeds at extracting a 3D shape and skeleton skinning weights which extrapolate well to poses not seen during training (out-of-distribution). However, the input TAVA needs to extract this info includes multi-view video as well as per-frame ground truth skeleton pose information, the latter of which might not be readily available to the common articulated objects. In addition, TAVA training time on a single input dataset currently can take up to 3-4 days even when using GPUs; this can be prohibitively long and prevent TAVA from being consistently used in some settings. To make the creation of 3D animatable avatars via TAVA more accessible, we plan to decrease TAVA's dependence on skeleton pose information as well as decrease TAVA's training time. Specifically, we aim to show that 1) approximate skeleton pose data can be inferred directly from multi-view video; 2) inferred skeleton pose data is sufficient to produce high quality extrapolations of 3D shape and skeleton skinning weights; and 3) TAVA is compatible with recent advances in NeRF training and can take advantage of this to yield results in reduced amounts of time. These contributions will more firmly place TAVA into the realm of inverse graphics for computer vision.

2 PRELIMINARIES

2.1 Rendering Neural Radiance Fields (NeRF)

NeRF [2] is a breakthrough technique for neural rendering of a *static* scene. It models the geometry and view-dependent appearance of the scene by using a multi-layer perceptron (MLP). Given a 3D coordinate $\mathbf{x} = (x, y, z)$ and the corresponding viewing direction (θ, ϕ), NeRF queries the emitted color $\mathbf{c} = (r, g, b)$ and material density σ at that location using the MLP. A pixel color $C(\mathbf{r})$ can then be computed by accumulating the view-dependent colors along the ray $\mathbf{r}$, weighted by their densities:

$$C(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(\sigma_i \delta_i))\mathbf{c}_i, \quad \text{where } T_i = \exp(-\sum_{j=1}^{i-1} \sigma_j \delta_j)\,, \tag{1}$$

where δ_i denotes the distances between the sample points along the ray. To further take the size of the pixels into consideration, Mip-NeRF [1] extends NeRF to represent each ray $\mathbf{r}$ that passes through a pixel as a cone, and the samples $\mathbf{x}$ along the ray as conical frusta, which can be modeled by multivariate Gaussians (μ, Σ). Thus, the density σ and view-dependent emitted color $\mathbf{c}$ for a sample on the ray are given by $F_\Theta : (\mu, \Sigma, \theta, \phi) \rightarrow (\mathbf{c}, \sigma)$, where $\mu = (x, y, z)$ is the

center of the Gaussian and $\Sigma \in \mathbb{R}^{3\times3}$ is its covariance matrix. The loss for optimizing the network parameters Θ of the neural radiance field is applied between the rendered pixel color $C(\mathbf{r})$ and the ground-truth $\hat{C}(\mathbf{r})$:

$$\mathcal{L}_{im} = \left\| C(\mathbf{r}) - \hat{C}(\mathbf{r}) \right\|_2^2 . \tag{2}$$

Please refer to the original papers [1, 2] for more details.

2.2 Template-free Animatable Volumetric Actors (TAVA)

TAVA represents an articulated subject as a volumetric neural actor in its canonical space. The representation includes a *Lambertian* neural radiance field F_{Θ_r} to represent the geometry and appearance of this actor, and a neural blend skinning function F_{Θ_s}, which describes how to animate the actor:

$$F_{\Theta_r} : (\mathbf{x}_c, \Sigma) \rightarrow (\mathbf{c}, \sigma), \qquad F_{\Theta_s} : \mathbf{x}_c \rightarrow \mathbf{w}, \tag{3}$$

where $\mathbf{c} = (r, g, b)$ is the material color, σ is the material density, and $\mathbf{w}$ is the skinning weights to blend all bone transformations for animation. Similar to Mip-NeRF [1], TAVA uses a multivariate Gaussian ($\mathbf{x}_c \in \mathbb{R}^3, \Sigma \in \mathbb{R}^{3\times3}$) to estimate the integral of samples within the volume of the discrete samples. Note that in most of the cases, an articulated actor is a Lambertian object, so TAVA excludes view directions from the input of F_{Θ_r}. With the skinning weights $\mathbf{w} = (w_1, w_2, ..., w_B, w_{bg}) \in \mathbb{R}^{B+1}$ defined in the canonical space, and given a pose $\mathbf{P} = \{\mathbf{T}_1, \mathbf{T}_2, ..., \mathbf{T}_B\} \in \mathbb{R}^{B\times4\times4}$, TAVA uses forward LBS to define the deformation of a point $\mathbf{x}_c$ in the canonical space to $\mathbf{x}_v$ in the view space:

$$\mathbf{x}_v = LBS(\mathbf{w}(\mathbf{x}_c; \Theta_s), \mathbf{P}, \mathbf{x}_c) = \left[\sum_{j=1}^{B} w_j(\mathbf{x}_c; \Theta_s) \cdot \mathbf{T}_j + w_{bg} \cdot \mathbf{I_d} \right] \mathbf{x}_c, \tag{4}$$

where $\mathbf{I}_d \in \mathbb{R}^{4\times4}$ is an identity matrix that allows the points in the background and empty space to *not* follow the skeleton when it is deformed.

3 POSE OPTIMIZATION

One of the reach goals of this project is to completely remove TAVA's reliance on per-frame ground-truth pose information, and instead learn the per-frame pose directly from 2D RGB images. Specifically, this would entail utilizing purely 2D multi-view images without any ground truth pose data to optimize for pose, skinning weights, shape, and appearance of an articulated subject. However, learning all these parameters simultaneously right off the bat is an extremely difficult task. Hence, to build up to this final goal, our approach is to first tackle two simplified versions of the problem: 1) Given noisy per-frame pose data, as well as RGB images alongside learned skinning weights, shape, and appearance, rectify the noisy per-frame pose; 2) Given no per-frame pose data, as well as RGB images alongside learned skinning weights, shape, and appearance, learn the per-frame pose. Once we are able to perform tasks 1 and 2 properly, we will then tackle the original problem: 3) Given RGB images only, learn per-frame pose, skinning weights, shape, and appearance.

3.1 Pose Model and Parameterization

In order to optimize per-frame pose in the TAVA pipeline, we must first parameterize it. For our current model, we define a pose as a collection of bones, which are 3D line segments drawn between the "head" and "tail" of the bone. Our current parameterization model uses a 7-tuple (a, b, c, d, x, y, z) to represent the position and orientation of a bone in a single pose and frame, where (a, b, c, d) is a quaternion representation of the direction of the vector drawn from the

bone head to the bone tail, and (x, y, z) is the position in world space of the bone head. Hence, if a single pose consists of n bones, and there are m frames in total, there will be $m \times n \times 7$ total parameters to describe per-frame poses.

From the original datasets used by TAVA, we can convert the per-frame ground truth poses into our parameterization model. For some of the subproblems we defined, we must be able to add noise to the set of ground-truth bones. To do so, we have opted for a noise model that is parameterized by two values: σ_{angle} and $\sigma_{translation}$. For each bone, we modify its orientation by independently adding some normally-distributed noise to each of its Euler angles, following the distribution $\mathcal{N}(0, \sigma_{angle})$. In addition, we modify its x-translation value by adding $\frac{1}{100} \times x_{bbox} \times \mathcal{N}(0, \sigma_{translation})$, where x_{bbox} is the x-direction size of the bounding box of the subject; we similarly do so for the y and z values. This model allows us to independently vary the degree of rotational and translational noise present in pose information.

3.2 Rectifying Noisy per-frame Poses

To focus on solely rectifying noisy pose, we first started out with a pre-trained TAVA model containing learned skinning weights, shape, and appearance for a subject. We then picked an arbitrary frame's pose, added noise to the pose, and applied back propagation to the pose parameters while freezing the rest of TAVA's model responsible for learning all other parameters. For this first model, the loss metric we used to drive the pose rectification process was purely the reconstruction loss defined in NeRF; hence, we will refer to this method as the "No Regularization" method.

To evaluate the performance of reconstruction, we also defined another loss function that measures the average L2 distance between the bone tail 3D positions in the rectified pose P and their corresponding bone tail 3D positions in the ground truth pose P^*:

$$\frac{1}{|P|} \sum_{b, b^* \in P, P^*} ||b - b^*||_2$$

We will refer to as bone tail loss; note that this loss is not used within training, but is a metric that gives insight into how well the pose reconstruction is performing.

For the first set of experiments we ran, we applied the No Regularization method to three different levels of perturbation to a hare skeleton. The first level, which we will refer to as "Small Perturbation", used noise parameters of $\sigma_{angle} = 8$ degrees and $\sigma_{translation} = 1$. The second level, which we will refer to as "Medium Perturbation", used noise parameters of $\sigma_{angle} = 16$ degrees and $\sigma_{translation} = 2$. The third level, which we will refer to as "Large Perturbation", used noise parameters of $\sigma_{angle} = 24$ degrees and $\sigma_{translation} = 3$. Visualizations of each noise level can be found in Figure 3, visualizations of the final pose optimization results after iteration 6000 can be found in Figure 4, and graphs of both NeRF reconstruction loss and bone tail loss from the experiment can be found in Figure 5.

From the loss statistics of the experiment, it seems that the No Regularization method is able to converge to a solution for all levels of pose noise, although the level of bone tail loss that is converged to increases with higher levels of initial injected noise. It is worth to note that the NeRF reconstruction loss, albeit noisy, also generally decreases over time, which is an encouraging sign for the overall goal of being able to optimize pose alongside other variables including shape and appearance.

At first glance, the reconstruction results after iteration 6000 also seem to match quite nicely with the ground truth pose. However, notice that in areas in which the pose optimization does not correctly capture the ground truth pose, the incorrect bone is often translated so that it lies in the general vicinity of the correct bone location, its orientation is incorrect. One conspicuous example of this is in the hare's front leg. Many "slightly off" bones also seem to be roughly

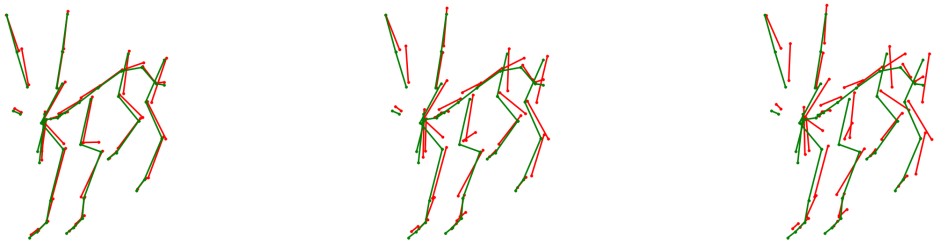

Fig. 3. Visualization of various amounts of noise added to a hare's ground truth pose. Ground truth pose is drawn in green, and perturbed pose is drawn in red. Small Perturbation is depicted on the left, Medium Perturbation is depicted in the middle, and Large Perturbation is depicted on the right.

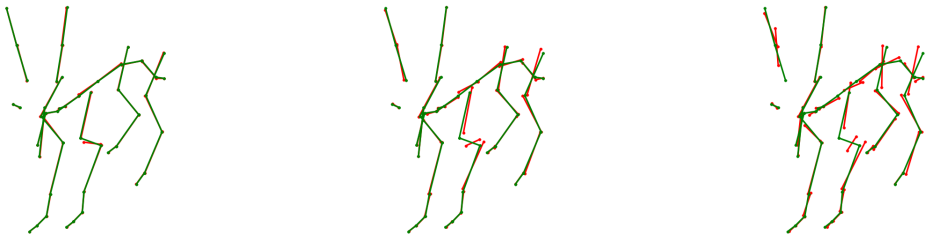

Fig. 4. Reconstruction results after iteration 6000 when running No Regularization method on various levels of perturbation. Ground truth pose is drawn in green, and perturbed pose is drawn in red. Small Perturbation is depicted on the left, Medium Perturbation is depicted in the middle, and Large Perturbation is depicted on the right.

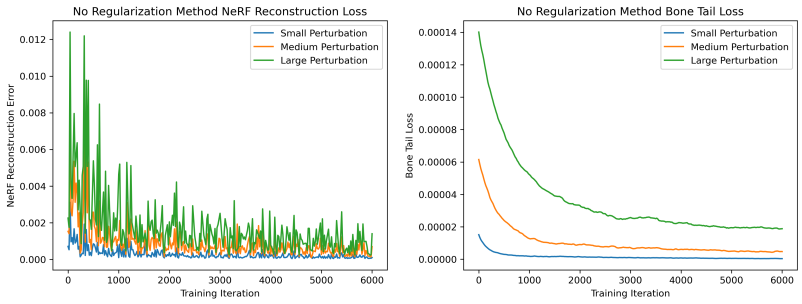

Fig. 5. Loss statistics for applying the No Regularization method on various levels of perturbation. NeRF reconstruction loss, which was the only loss used in this method, is depicted on the left. Bone tail loss is depicted on the right.

parallel with their original perturbed orientations, which suggests that the current model may find it hard to rotate bones.

To test this intuition, we performed a second test in which we applied the No Regularization method to two new types of perturbed pose input. The first type, which we will refer to as "Pure Rotation Perturbation", used noise parameters of $\sigma_{angle} = 24$ degrees and $\sigma_{translation} = 0$; on the other hand, the second type, which we will refer to as "Pure Translational Perturbation", used noise parameters of $\sigma_{angle} = 0$ degrees and $\sigma_{translation} = 3$. Note that these two new types of perturbations are essentially separating the rotational and translational perturbations performed by Large Perturbation. The two new types of perturbed noise input, as well as Large Perturbation, can be seen in Figure 6. Visualizations of the final pose optimization results after iteration 6000 can be found in Figure 7, and graphs of both NeRF reconstruction loss and bone tail loss from the experiment can be found in Figure 8.

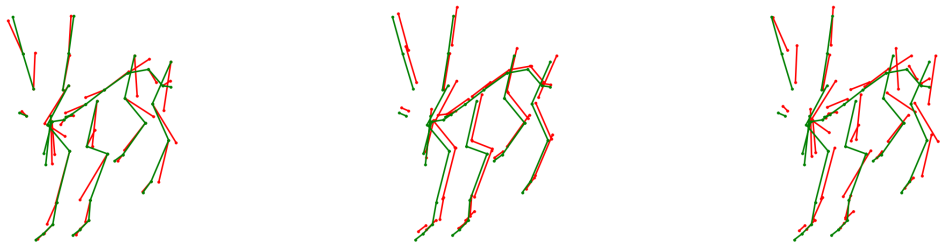

Fig. 6. Visualization of Pure Rotational Perturbation (left), Pure Translational Perturbation (middle), and Large Perturbation (right), which is the net effect of the first two perturbations. Ground truth pose is drawn in green, and perturbed pose is drawn in red.

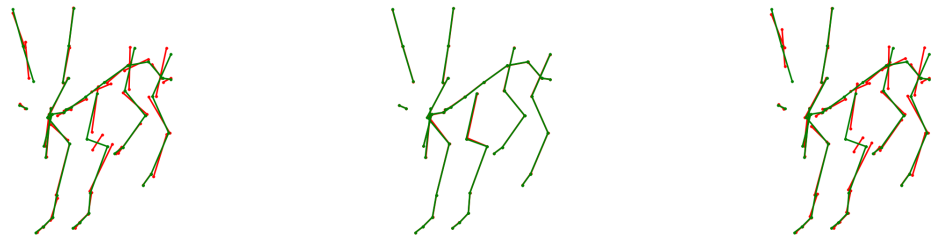

Fig. 7. Reconstruction results after iteration 6000 when running No Regularization method on Pure Rotational Perturbation (left), Pure Translational Perturbation (middle), and Large Perturbation (right). Ground truth pose is drawn in green, and perturbed pose is drawn in red.

From the loss statistics of the experiment, it seems that the No Regularization method is able to converge to an almost perfect solution for pure translational pose noise in terms of NeRF reconstruction loss and bone tail loss, but

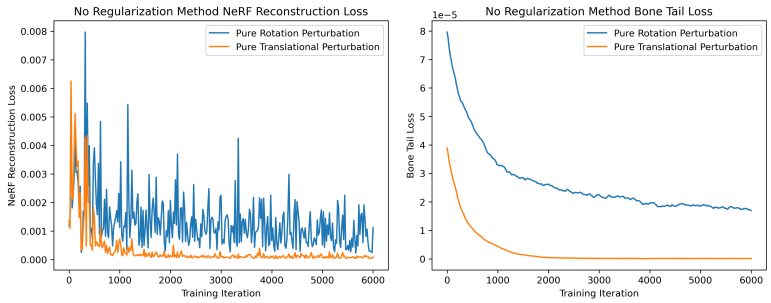

Fig. 8. Loss statistics for applying the No Regularization method on Pure Rotational Perturbation and Pure Translational Perturbation. NeRF reconstruction loss, which was the only loss used in this method, is depicted on the left. Bone tail loss is depicted on the right.

struggles much more with pure rotational pose noise. This can be visually confirmed in the actual iteration 6000 pose optimization results as well. The result for Pure Translational Perturbation seems to overlap with the ground truth pose, whereas the pose result from Pure Rotational Perturbation seems to exhibit many of the same issues that the pose result from Large Perturbation has.

From these results, we are currently thinking of two different solutions to help the system better learn bone rotations. The first idea is to add an additional regularization term to the NeRF reconstruction loss during pose optimization training, such that this term encourages bones that are connected in the canonical pose to stay close to each other during optimization. The second idea is to reparameterize the bones so that the head of each bone must be connected to the tail of its "parent" bone according to the canonical pose. This could potentially lead to more physically-viable poses to be generated, and would make the parameters associated with most bones to be purely rotational. Quaternion representation of the orientation of the bone may not be very easy to optimize over, so we could consider using another representation, such as Eulerian angles.

4 CUDA OPTIMIZATIONS FOR RENDERING

In this section, we describe how we optimize the rendering procedure to better utilize the GPU. We write custom CUDA kernels for (1) sampling points along rays and for (2) volume rendering of rays with a variable-number of samples per ray. We compare the original PyTorch implementation with our custom CUDA implementation, and we report timings. We use this section to describe what CUDA is and how it works, as well as how we are approaching our problem for speedup gains.

4.1 Why do we want to use CUDA kernels?

An obvious question is, "Why do we need CUDA kernels?" as opposed to simply leveraging code such as PyTorch, which already has CUDA implementations. The main issue is that PyTorch is often built for general purpose utilities, and our ray sampling and rendering procedure is rather specific. PyTorch is best for batch operations where the dimensions of inputs are fixed. However, we wish to render rays with a variable number of samples because the rays are variable length in 3D space. For this reason, we can leverage CUDA kernels to process rays independently–one per GPU thread, which is highly parallelizable because GPUs have thousands of threads.

Variable-number of samples per ray. Currently the TAVA codebase assumes that each ray–shooting from a camera origin into the scene–will intersect with the human at two locations, t_{near} and t_{far}, and that the number of

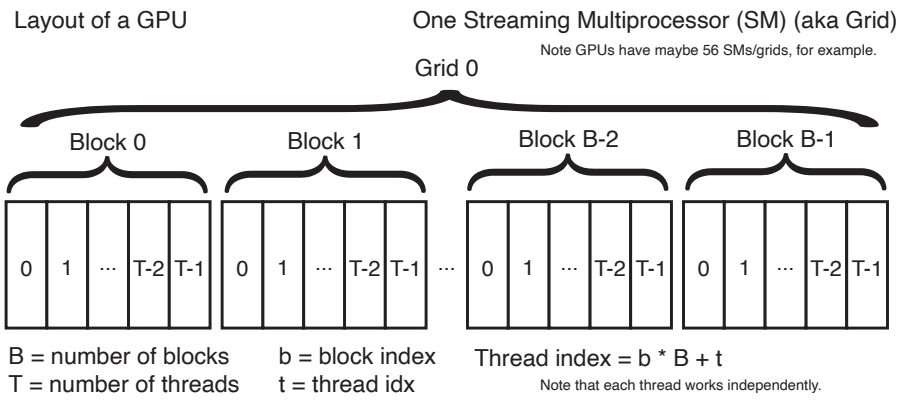

Fig. 9. **Layout of a GPU.** Here is a layout depiction of a GPU. It's composed of many Streaming Multiprocessors, which in turn have multiple blocks, and the blocks have multiple threads. Each thread does work and can be used to access indices of tensors and perform operations. Notice the "Thread idx = " equation indicating how to figure out where on the GPU you are operating. This is important notation because this is how CUDA kernels will know what part of the tensor is being operated on (hence, why it's parallelizable). See this resource for a similar picture and more details: https://developer.nvidia.com/blog/even-easier-introduction-cuda/.

samples is fixed (e.g., 64 samples per ray). This is easy to implement with PyTorch for a fixed number of samples because batches of rays will be processed into tensors of shape (number of rays, number of samples, 3), where 3 denotes the an xyz position in space. However, as we show in Figure 10, this is problematic for efficiency when $t_{dist} = t_{far} - t_{near}$ is not equal for each ray. In other words, some regions of space will be oversampled and others will be undersampled.

4.2 How does a GPU and CUDA work?

A GPU (Graphics Processing Unit) can has many threads that can be used in parallel, which makes it very fast for matrix multiplication, ray tracing, and other operations. In practice, we can program a GPU with CUDA code that is compiled with NVIDIA's CUDA Compiler (NVCC). It works by having CPU code make calls to the GPU. First, data has to be in the "Allocate Unified Memory", which is accessible to both the CPU and GPU. NVCC compiles the CUDA code, which lives in ".cu" files. CUDA GPUs run kernels with numbers of threads that are multiples of 32 (e.g., 1024). Kernels have access to the indices of their running threads. This is important when processing a tensor. A block of parallel threads is called a grid. If number of elements (e.g., rays) is greater than the grid size (total number of threads per grid), then you have to have a for loop adding the "grid dim" as an offset. As mentioned in Figure 9, the thread index can be calculated based on the block index, block size, and thread index inside the kernel to inform where the processing is occurring on the input tensor.

4.3 Handling different number of samples per ray

. We can use a CUDA kernel to process each ray independently for ray sampling, shown on the left of Figure 10. We implement a kernel function called *sample_uniformly_along_ray_kernel* which will take in a ray, near and far values, and a sample size, and it will compute the positions along the ray. Then, we take all the samples and can bring them back into PyTorch land to run a forward pass with our MLP. Finally, we can use another code kernel (right of Figure 10) to render rays with variable length. However, this is more complicated because it requires implementing a backward pass and explicitly computing the gradient. *We will try to finish by the end of the class, but we may fall short of completing*

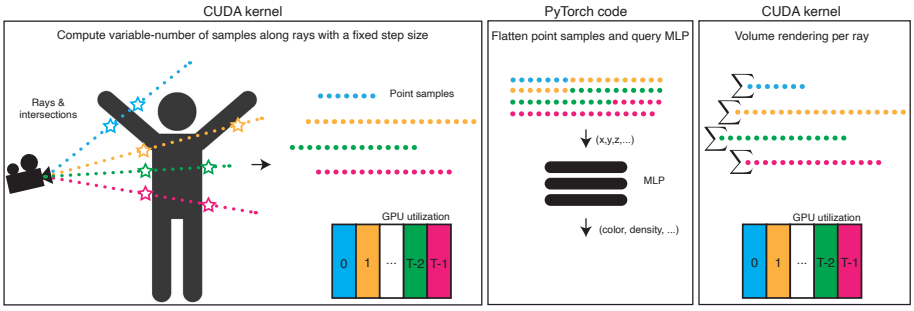

Fig. 10. **Our CUDA optimizations.** Here is the framework depicting where we add CUDA optimizations. We can use CUDA kernels for ray sampling (left) and also for rendering (right). In the middle, we can simply use PyTorch because the MLP operates on each point independently, so there is no variable-size inputs to deal with. This also leads to clean code for a user than can easily be placed into any NeRF implementation.

this kernel. Note that we use a different GPU thread for each ray, which leads to high throughput and eliminates oversampling/undersampling issues in the current TAVA framework.

For intuition on why variable length inputs are important, consider the blue ray and the pink ray in Figure 10. Clearly they should have different number of samples per ray, but with PyTorch code it's much easier to simply have the same number of samples and use a function call such as *linspace* with "number of samples" (e.g., 64) for the given batch.

4.4 Profiling code experiments.

Debugging CUDA is difficult due to it's parallelized nature, but we will be running baselines on PyTorch-only code vs. our CUDA kernel versions. We are using pybind11 to create a nice abstract Python class that implements *nn.Module* so that our optimizations are plug-and-play with TAVA's existing codebase with minimal changes.

Table 1. **Code profiling.** Here we report PyTorch speed vs. our CUDA code with 1024 rays with 64 samples each, with varying lengths, i.e., $t_{dist} = t_{max} - t_{min}$ are different.

Code	Ray sampling (sec / ray)	Volume rendering (sec / ray)
PyTorch only		
With CUDA kernel		

ACKNOWLEDGMENTS

We'd like to thank the TAs for feedback on the project proposal and throughout the project.

REFERENCES

[1] Jonathan T Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P Srinivasan. 2021. Mip-nerf: A multiscale representation for anti-aliasing neural radiance fields. In *International Conference on Computer Vision*.

[2] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2020. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*. Springer, 405–421.