Final Project Paper

TensoRF Improvements Using Custom CUDA Kernel

By: Gautam Mahesh, Venkata Kishore Marisetty

Model Overview

Background:

TensoRF is an extension model building off of the state of the art novel visual synthesis method NeRF, which seeks to improve on the efficiency as well as rendering quality compared to the previous model. This is accomplished by instead of relying on Multi Layer Perceptrons for representations of a 3D scene, it is instead reliant on a voxel grid approach composed of factorized tensors. This scene representation is simplified by breaking down the scene into sets of simple elements of matrices and vectors that effectively model the scene. In this case the sets are the xy plane and z line, the xz plane and y line, and the yz plane and x line. These effectively represent a scene without the increased complexity of an MLP, providing the backbone of the improvements provided by TensoRF.

Model Architecture:

In our case, this project focused on the TensorVMSplit architecture, which is comprised of the following:

- Voxel Grid representation: Composed of sets of plane line tensors utilizing tensor decomposition. Utilizing this scene representation will provide density (opacity) and color features per sampled point of the scene.
- Small MLP decoder: This step in the model is meant to process appearance features (opacity and color) from all sampled points to produce the final colors to be rendered in the following step. The architecture of the MLP is as follows:

```
MLPRender_Fea(
  (mlp): Sequential(
    (0): Linear(in_features=150, out_features=128, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=128, out_features=3, bias=True)
  )
)
```

- Volume Rendering: This is taken directly from NeRF, where it uses the standard volume rendering equation that takes in the color values from the Multi-Layer Perceptron to produce the final rendered image

Inputs and Outputs

For the inputs of this model, this is similar to NeRF where scene sets either surrounding the key object or taken from a set angle are fed into the model to be trained on. The model can only be trained on one scene at a time, with it having no general capabilities to learn and render multiple scenes. For the outputs of the model, these are renders of the scene taken from explicitly defined camera points surrounding the object of interest.

In our case, we experimented heavily with the nerf-synthetic dataset, which is a collection of multiple scenes, each composed of about 100 images taken from viewpoints on the upper hemisphere surrounding the focus object. These objects include a lego model, a truck, a flower, a wine holder and more. For this project, we focused entirely on one scene due to the time constraints of the project.

Training/Render Process:

For training the model, we specified that it would run for 15000 iterations with a batch size of 2046. These were chosen based on the limitations of the working environment, which ran into memory issues if trained on the recommended batch size of 4096. For training, this used photometric mean squared error loss to judge the render quality while training and utilized the Adam optimizer while training.

For rendering, this can either be done in conjunction with training, or be done on its own given a pretrained model on a scene.

Project Effort Overview

For optimizing the TensoRF model, the following stages were proposed:

- Baseline evaluation of the model for both training and inference runs.
- In depth profiling of both runs to identify hot and cold spots
- CUDA kernel development targeting hot spots identified by profiling TensoRF
- Iterative improvement of kernel design based on results of previous runs with the CUDA kernel implementation

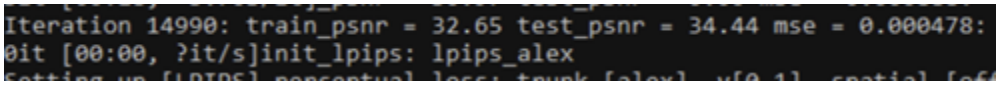
As stated before, the scene used for testing was the Synthetic-NeRF lego scene, with the training run configured to do 15k iterations with a batch size of 2048. For inference,

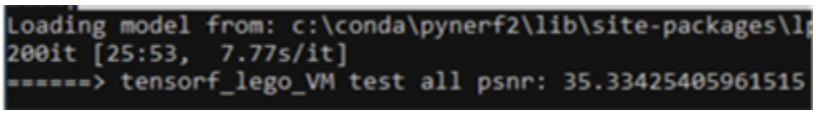
the total run would be for rendering 200 images, and validating both the runtime in total, the runtime per image, as well as the visual image render quality.

Experimentation

Baseline Results:

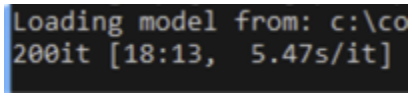
For training, after 15k iterations with a batch size of 2048, we received the following metrics for training and render run:





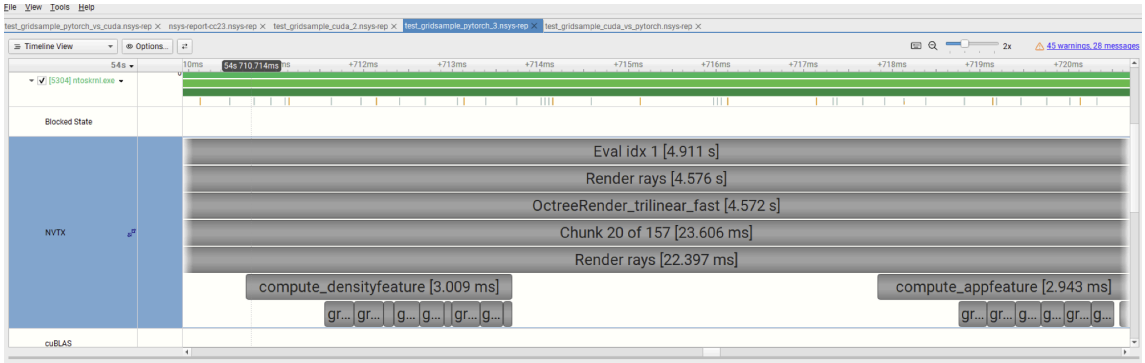
Compared against the TensoRF paper, the PSNR scores aligned with the expected PSNR, with the final test PSNR score exceeding what was recorded on the source paper. This is likely due to the scores within the paper being the average across all scenes within the synthetic-nerf dataset, resulting in a lower average compared to our run where it was trained against one scene.

For the inference only run, the following results were retrieved:



Profiling:

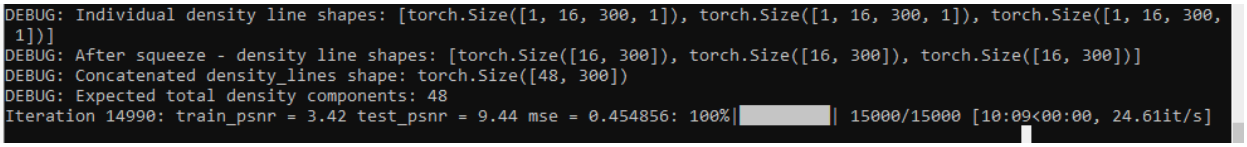
When profiling both the training and inference runs the most prominent section that took up a majority of the execution time per iteration was the OctreeRender_trilinear_fast function. This function is responsible for taking sampled points from the 3D scene, and retrieving the appropriate features (color and opacity) for all points sampled from the scene as well as rendering the full scene.



From this trace, per iteration most of the time within each iteration is dedicated to the operations within the function, making it a prime candidate for optimization.

First Attempt (Failed):

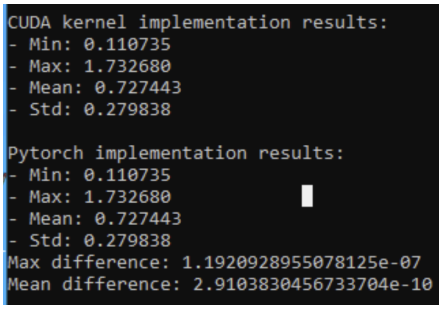
Based on our initial profiling analysis we jumped on trying to convert OctreeRender_trilenear_fast pytorch implementation to CUDA extension, we created fused ray renderer cuda kernel that would directly takes batch of rays as input and produces density and alpha(pixel color), after handling couple of runtime errors we managed to make this kernel working for training; but the correctness of the kernel seem to be suffered alot and reliability was also hit during updateAplha and upsampling steps we got very low PSNR values and sigma feature values and mean squared error reaching 0 values.
Low PSNR after training with fused_ray_render_kernel



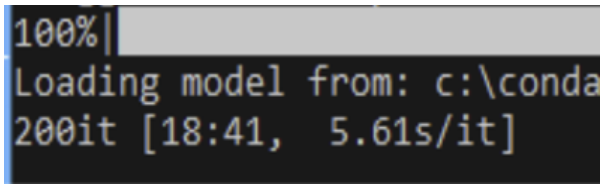
We analysed that our kennel doesn't handle gradient and backpropagation computations not correctly done and fixing them is hard for the current project timeline to tackle, So we pivoted our focus into starting for only render task which was originally taking ~18.30 mins for the lego scene.

Second Attempt:

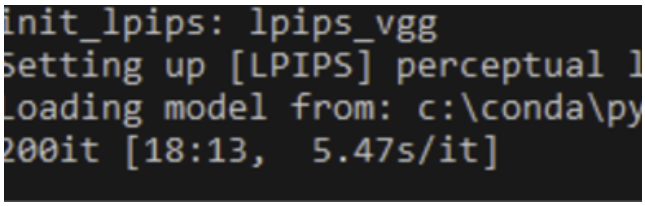
For this attempt, instead of tackling the whole process of multiple grid_samples as well as the multiple sets of matrix vector combinations, we instead focused on creating a kernel replicating the grid_sample function, which was responsible for interpolation of all sampled points given either a plane or line as well as an image. To test this functionality before integrating with the full model, a test harness was created to compare against the pytorch grid_sample function and our CUDA kernel implementation. This test harness is present under the test.py file in the repository. From testing the kernel implementation against the baseline, the below image shows minimal difference between the two methods.



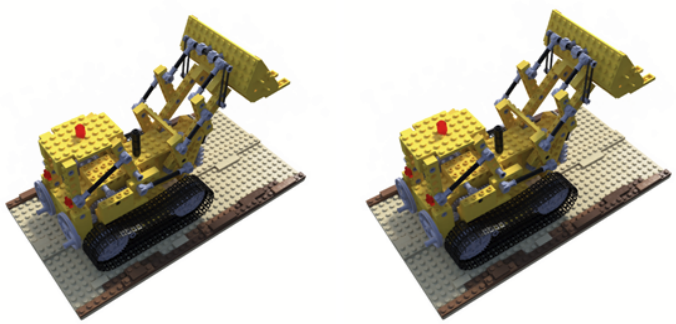
Given the difference in values between the two methods, we were confident in incorporating this CUDA kernel into the main TensoRF run. An additional consideration we made when designing the kernel is that this implementation would only focus on inference run due to the complicated nature of incorporating backpropagation within the CUDA kernel, which is necessary for successful training. After incorporating the kernel within the compute_densityfeature function, replacing the grid_sample pytorch function call, we received the following results.



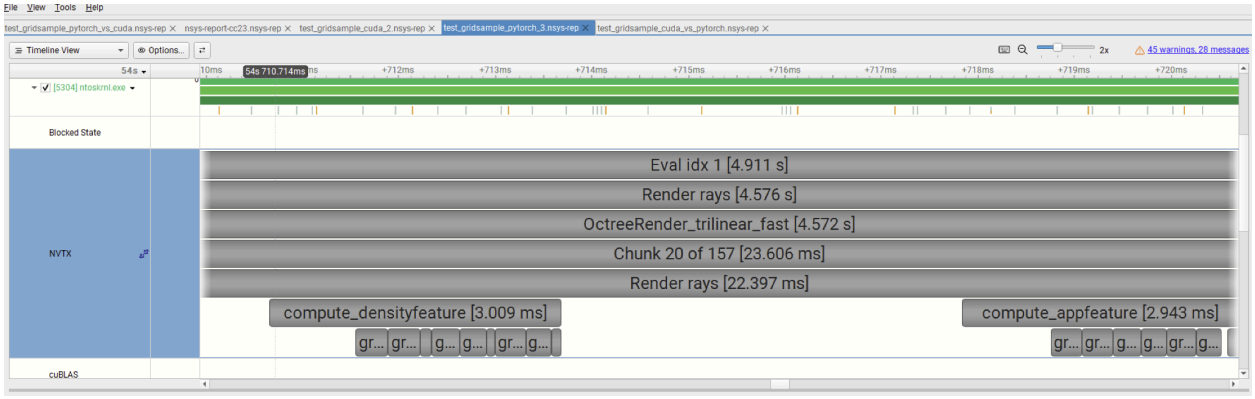
Compared against the baseline, however, this was not a noticeable improvement, taking the same amount of time to complete as the standard pytorch implementation, which had the following results rendering 200 images.



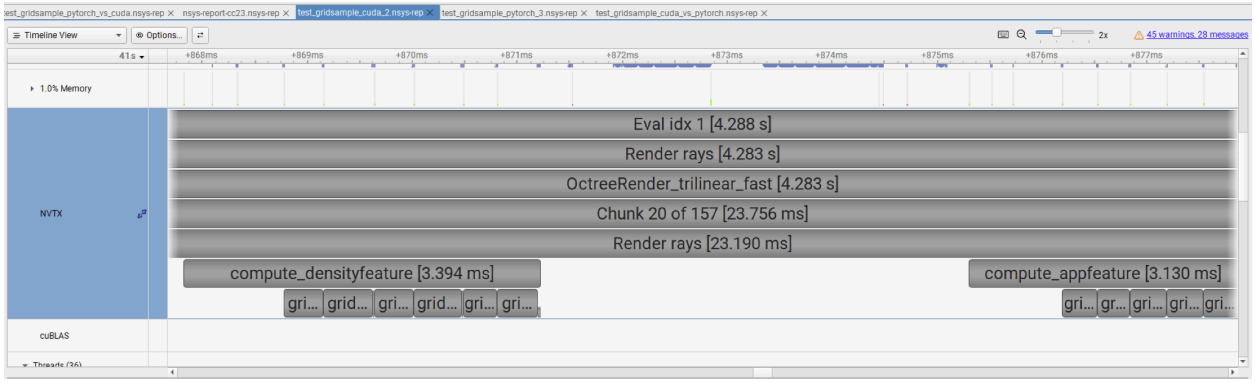
When looking at the renders between the two images there was also minimal difference between the two inference runs, each having minimal visual artifacts and being indistinct to one another when the same view was utilized. The below images show the baseline render on the left and the CUDA render on the right.



When profiling these two implementations, we found that comparing the grid_sample calls and the CUDA calls were almost identical in length:



Pytorch trace



CUDA Kernel Trace

From these profiling traces, we were able to determine that just transferring the grid_sample functionality over to CUDA was not enough and that it was already heavily optimized. As a result, with the next iteration, we decided to encapsulate all plane line set density feature calculations within the calculation to move sequential code to parallel processes.

Third attempt:

After successful render results from our *grid_samples* kernel and analysing the overall render time improvements are lower we decided to address multiple kernel launches happening for each plane and each line during compute_density feature, we created a new fused kernel - fused_plane_line_single.

This kernel computes the fused product of features sampled from a 2D plane and a 1D line for each input coordinate, combining them per channel. It interpolates values from the plane and line using bilinear and linear interpolation, respectively, and accumulates their product for each sample.

We harness the unit testing mechanism to write python script to validate kernel correctness with numerical agreement and then integrated into render pipeline.

The result is atomically added to the output tensor to ensure thread safety during parallel execution.

Result:
We have observed a good amount of gains with respect to time taken for each iteration reduced by ~36% and we don't observe any quality reduction of the rendered output.

Unit test output:

```
=== Testing basic functionality ===
Fused CUDA kernel time: 0.001027 seconds
PyTorch reference time: 0.812083 seconds
Speedup: 790.83x

Results comparison:
  Max difference: 0.000001
  Mean difference: 0.000000
  Relative error: 0.000000

=== Testing edge cases ===

Testing zeros:
  boundary: ✓ (NaN: False, Inf: False)
  beyond: ✓ (NaN: False, Inf: False)
  random: ✓ (NaN: False, Inf: False)

Testing ones:
  boundary: ✓ (NaN: False, Inf: False)
  beyond: ✓ (NaN: False, Inf: False)
  random: ✓ (NaN: False, Inf: False)

Testing large_values:
  boundary: ✓ (NaN: False, Inf: False)
  beyond: ✓ (NaN: False, Inf: False)
  random: ✓ (NaN: False, Inf: False)

Testing small_values:
  boundary: ✓ (NaN: False, Inf: False)
  beyond: ✓ (NaN: False, Inf: False)
  random: ✓ (NaN: False, Inf: False)

=== Testing model integration simulation ===
Testing with 2048 points, grid size 64, 8 components
Integration simulation results:
  CUDA time: 0.000101s, PyTorch time: 0.000160s
  Speedup: 1.58x
  Max difference: 0.000002
  Mean difference: 0.000000
  Relative error: 0.000000
  Results match:  ✓
```

Render output:

TensoRF - Pytorch

TensoRF - Pytorch+CUDA (ours)

Render time output:

```
sampling step size:  tensor(0.0029, device='cuda:0')
sampling number:  1102
pos_pe 6 view_pe 2 fea_pe 2
MLPRender_Fea(
  (mlp): Sequential(
    (0): Linear(in_features=150, out_features=128, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ReLU(inplace=True)
    (4): Linear(in_features=128, out_features=3, bias=True)
  )
)
200it [11:33,  3.47s/it]
TensorVMSplit densityfeature calls: 24897
TensorVMSplit appfeature calls: 23976
```

Conclusion & Next Steps

In this project, we explored multiple CUDA kernel optimizations to improve the inference performance of the TensoRF model. Through careful profiling, we identified the `OctreeRender_trilinear_fast` and `grid_sample` functions as key performance bottlenecks. Our first attempt to fully replace the ray rendering and feature computation with a fused CUDA kernel revealed the complexity of handling gradients and backpropagation, leading us to pivot towards optimizing the inference-only path.

Our second attempt involved implementing a custom CUDA version of the `grid_sample` function, which, while functionally correct and verified through unit tests, did not yield noticeable performance gains compared to PyTorch's already highly optimized implementation.

In our third and most successful attempt, we designed and integrated the `fused_plane_line_single` kernel, which fused the computation of features from 2D planes and 1D lines into a single CUDA kernel. This reduced redundant kernel launches and improved memory access patterns, resulting in an approximate **36% reduction in per-iteration render time** without compromising render quality.

Method	Render Time per 200 images	Relative Speedup	Render Quality Impact
Baseline (PyTorch)	18.30 min	1.00× (baseline)	None
CUDA grid_sample (2nd Attempt)	18.25 min	~1.00× (no gain)	None
Fused Plane-Line Kernel (3rd Attempt)	**11.71 min**	**1.56× (~36% faster)**	None

Learnings:

- Profiling-driven development is critical; assumptions without evidence can lead to wasted effort.
- Unit testing CUDA kernels before model integration ensures correctness and saves debugging time.
- Handling backpropagation in CUDA extensions requires deep understanding of PyTorch's autograd system and was beyond the scope of this project.
- Inference-only CUDA optimizations offer an effective and manageable first step for real performance improvement.

Next Steps:

- **Extend CUDA optimization to the ray rendering and batch processing stages** to reduce kernel launch overhead and improve memory coalescing, which could provide further performance gains.

- Explore integrating custom CUDA kernels for the **training path**, including gradient computation, for end-to-end optimization.

- Investigate advanced techniques such as **persistent threads, warp-level primitives, and asynchronous memory transfers** to further boost performance.

- Apply these optimizations to multi-scene training or real-world datasets to evaluate scalability and generalization of the improvements.