Neural Radiance Fields to Implementation

IF YOU ONLY KNEW THE POWER OF THE DARK SIDE

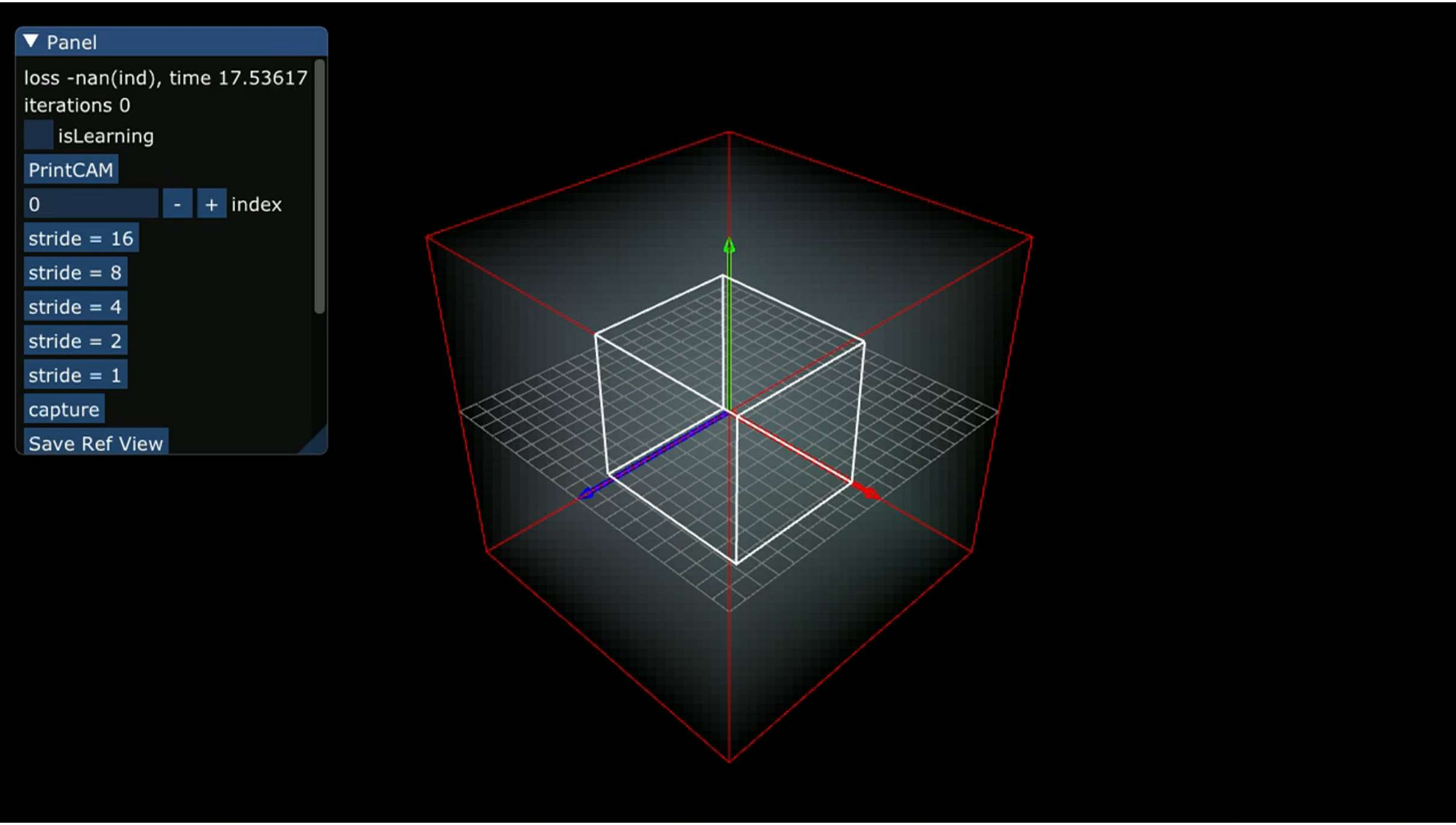
Ushio

What's Neural Radiance Fields?

- A novel volumetric scene representation
  - Location(X, Y, Z) and Direction(θ, ϕ) to Radiance and volume density (R, G, B, σ)
  - Neural-based volume encoding – fully-connected neural network
- Constructed from sparse reference inputs such as photos
- View synthesizing via volume ray marching

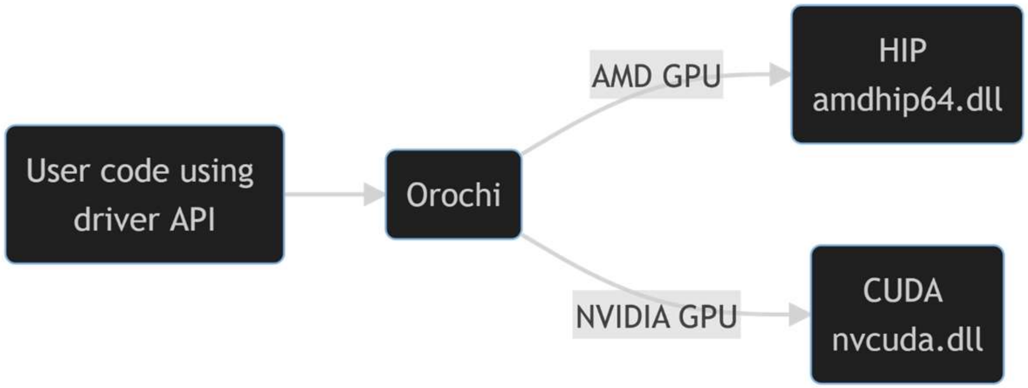
Inputs


r_0.png
r_1.png
r_2.png
r_3.png
r_4.png
r_5.png
r_6.png
r_7.png
r_8.png
r_9.png
r_10.png
r_11.png
r_12.png
r_13.png
r_14.png
r_15.png
r_16.png
r_17.png
r_18.png
r_19.png
r_20.png
r_21.png
r_22.png
r_23.png
r_24.png
r_25.png
r_26.png
r_27.png
r_28.png
r_29.png
r_30.png
r_31.png
r_32.png
r_33.png
r_34.png
r_35.png
r_36.png
r_37.png
r_38.png
r_39.png
r_40.png
r_41.png
r_42.png
r_43.png
r_44.png
r_45.png
r_46.png
r_47.png


Panel
loss -nan(ind), time 17.53617
iterations 0
isLearning
PrintCAM
0
-
+
index
stride = 16
stride = 8
stride = 4
stride = 2
stride = 1
capture
Save Ref View

Environment

- Radeon Pro W6800 GPU
- CUDA (HIP – via Orochi)
- Windows
- Works also on NV GPU

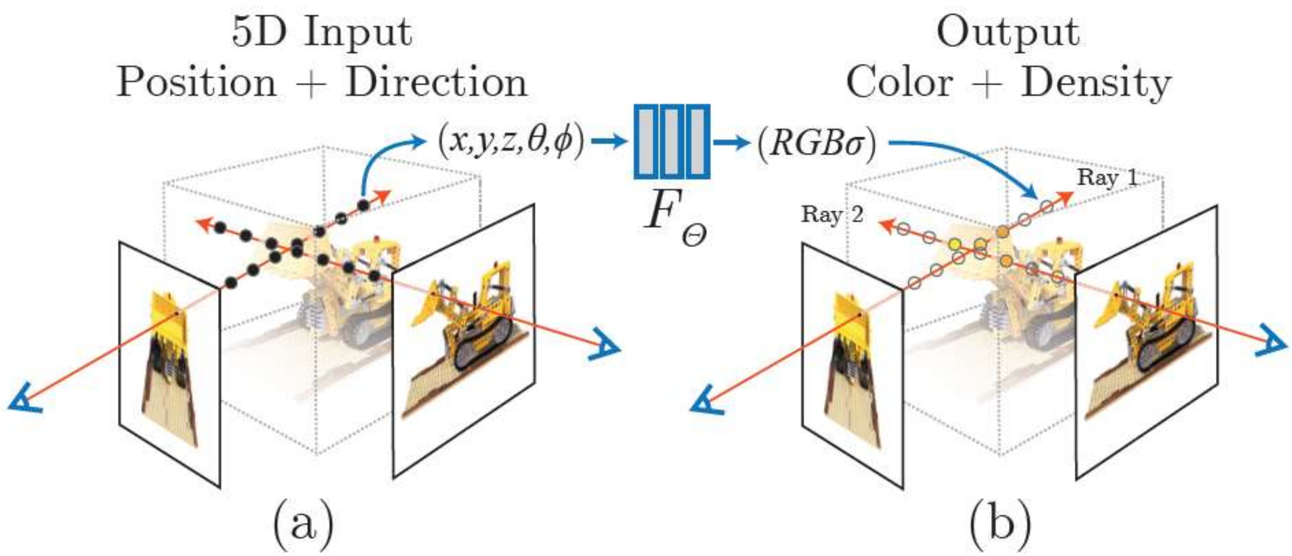


https://gpuopen.com/learn/introducing-orochi/
https://github.com/GPUOpen-LibrariesAndSDKs/Orochi

Deep dive into NeRF

Scene Representation as Volume

- A fancy function F - Location(X, Y, Z) and Direction(θ, ϕ) to Radiance and volume density (R, G, B, σ)
- F is a fully-connected neural network!



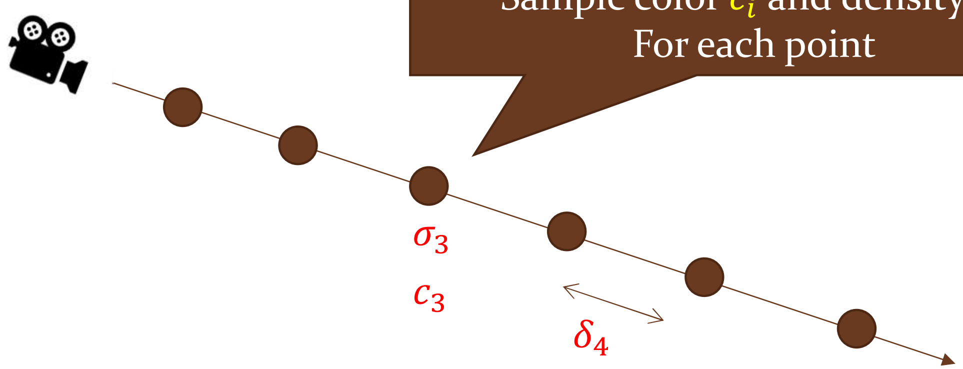
NeRF paper - Fig. 2

Volume Rendering

- Ray marching sample generation
  - Arbitrary sample strategy is applicable
    - Uniform
    - Proportional to a distance from its camera
    - NDC(normalized device coordinates)

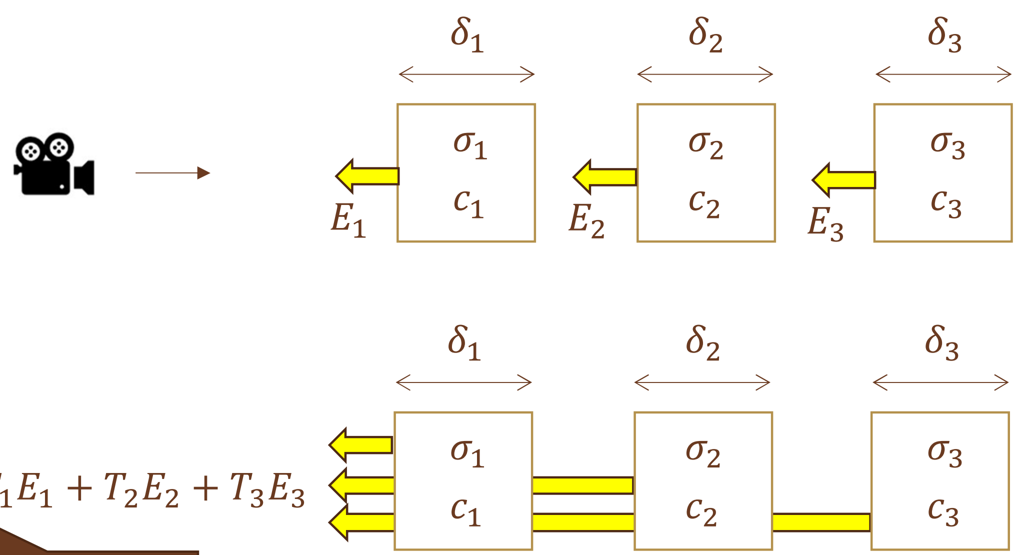
Still not sure what's the best
So let me skip this topic for now

Sample color c_i and density σ_i
For each point



Volume Rendering

- Each subvolume emits some radiance & absorbed by the front subvolume



$$\hat{C} = T_1E_1 + T_2E_2 + T_3E_3$$

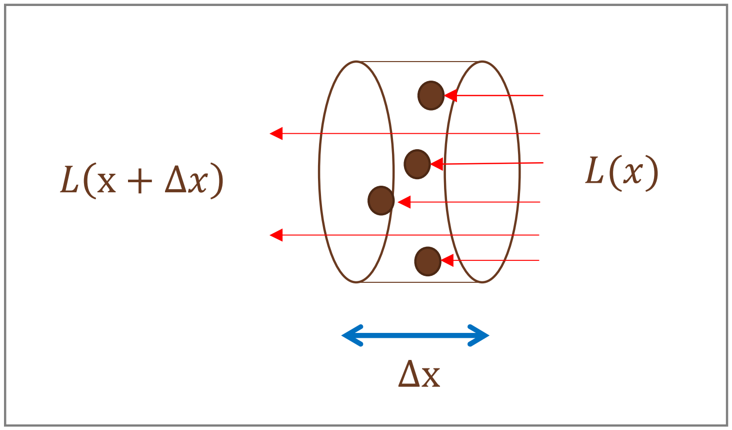
Absorption T_i

Absorption

$$\frac{L(\mathrm{x}+\Delta x)-L(\mathrm{x})}{\Delta x}=-\sigma L(\mathrm{x})$$

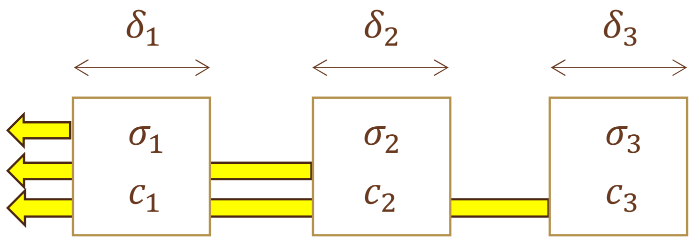
$$\frac{dL(x)}{dx}=-\sigma L(x)$$

$$L(\mathrm{x})=\exp(-\sigma x)L(0)$$



Absorption

$$\hat{C} = T_1E_1 + T_2E_2 + T_3E_3$$

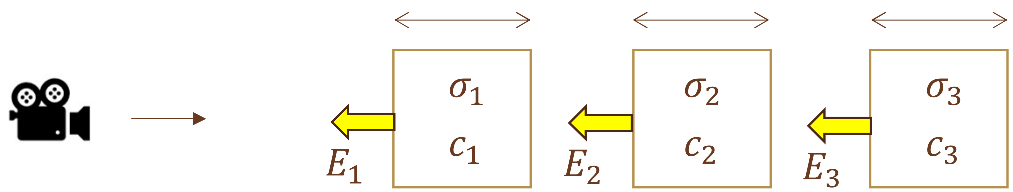


$$T_1 = 1$$
$$T_2 = \exp(-\sigma_1\delta_1)$$
$$T_3 = \exp(-\sigma_1\delta_1)\exp(-\sigma_2\delta_2) = \exp(-\sigma_1\delta_1 - \sigma_2\delta_2)$$

$$T_i = \prod_{j=1}^{i-1} \exp(-\sigma_i\delta_i) = \exp(-\sum_{j=1}^{i-1} \sigma_i\delta_i)$$

Emission

- What about E_i?

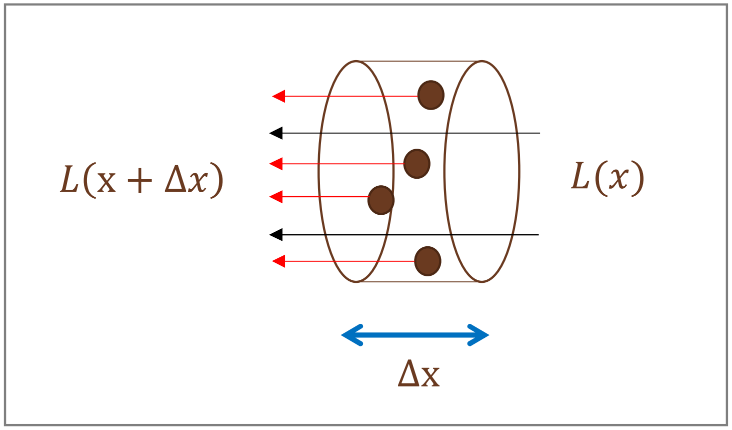


Emission

$$\frac{E(\mathrm{x}+\Delta x)-E(\mathrm{x})}{\Delta x}=\sigma c$$

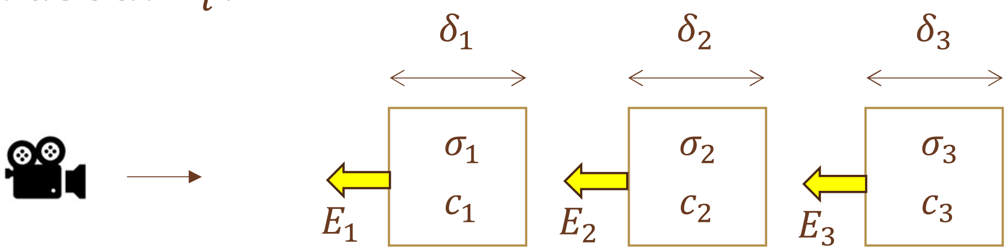
$$\Downarrow$$

$$
\begin{aligned}
E &= \int_0^x \sigma c \exp(-\sigma x)dx \\
&= \sigma c \int_0^x \exp(-\sigma x)dx \\
&= \sigma c\left(\frac{1-\exp(-\sigma x)}{\sigma}\right) \\
&= c(1-\exp(-\sigma x))
\end{aligned}
$$



Emission

- What about E_i?



$$E_1 = c_1(1 - \exp(-\sigma_1 \delta_1))$$
$$E_2 = c_2(1 - \exp(-\sigma_2 \delta_2))$$
$$E_3 = c_3(1 - \exp(-\sigma_3 \delta_3))$$

$$\Downarrow$$

$$E_i = c_i(1 - \exp(-\sigma_i \delta_i))$$

Volume Rendering

- The final form

$$\hat{C} = \sum_{i=1}^{N} T_i \underbrace{(1 - \exp(-\sigma_i \delta_i)) c_i}_{\text{Emission}} \qquad \underbrace{T_i = \exp(-\sum_{i=1}^{i-1} \sigma_i \delta_i)}_{\text{Absorption}}$$

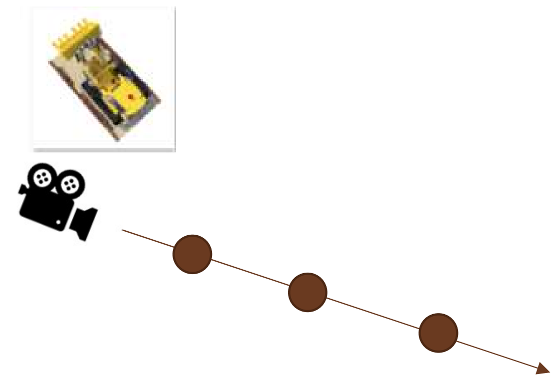
NeRF paper, Eq .3

results in the MLP being evaluated at continuous positions over the course of optimization. We use these samples to estimate $C(\mathbf{r})$ with the quadrature rule discussed in the volume rendering review by Max [26]:

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i (1 - \exp(-\sigma_i \delta_i)) \mathbf{c}_i \,, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right) , \qquad (3)$$

where $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples. This function for calculating $\hat{C}(\mathbf{r})$ from the set of $(\mathbf{c}_i, \sigma_i)$ values is trivially differentiable and

Training



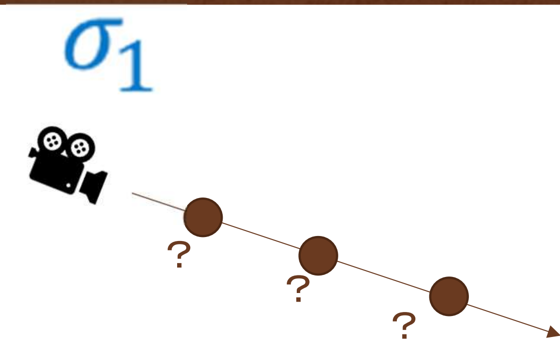
- Loss function $\mathcal{L}$ is based on the color difference

Example) Simple squared error:

$$\mathcal{L} = \frac{1}{2}\left(\hat{C}_r - C_r\right)^2 + \frac{1}{2}\left(\hat{C}_g - C_g\right)^2 + \frac{1}{2}\left(\hat{C}_b - C_b\right)^2$$

$$\frac{d\mathcal{L}}{d\hat{C}_r} = \hat{C}_r - C_r \qquad \frac{d\mathcal{L}}{d\hat{C}_g} = \hat{C}_g - C_g \qquad \frac{d\mathcal{L}}{d\hat{C}_b} = \hat{C}_b - C_b$$

Remark:
Instant NGP repository uses "Huber Loss" and it is more stable and faster to converge.



Training

- We need all derivatives of the learnable parameters
  - However, we only know $\frac{d\mathcal{L}}{d\hat{C}_r}, \frac{d\mathcal{L}}{d\hat{C}_g}, \frac{d\mathcal{L}}{d\hat{C}_b}$, not the derivatives of these parameters

Known derivatives

$\frac{d\mathcal{L}}{d\hat{C}_r}, \frac{d\mathcal{L}}{d\hat{C}_g}, \frac{d\mathcal{L}}{d\hat{C}_b}$

δ_1	δ_2	δ_3
σ_1, c_1	σ_2, c_2	σ_3, c_3
$\frac{d\mathcal{L}}{d\sigma_1}$	$\frac{d\mathcal{L}}{d\sigma_2}$	$\frac{d\mathcal{L}}{d\sigma_3}$
$\frac{d\mathcal{L}}{dc_1}$	$\frac{d\mathcal{L}}{dc_2}$	$\frac{d\mathcal{L}}{dc_3}$

Then, you can apply chain rule

$$\frac{d\mathcal{L}}{d\sigma_i} = \boxed{\frac{d\hat{C}_r}{d\sigma_i}}\frac{d\mathcal{L}}{d\hat{C}_r} + \boxed{\frac{d\hat{C}_g}{d\sigma_i}}\frac{d\mathcal{L}}{d\hat{C}_g} + \boxed{\frac{d\hat{C}_b}{d\sigma_i}}\frac{d\mathcal{L}}{d\hat{C}_b}$$

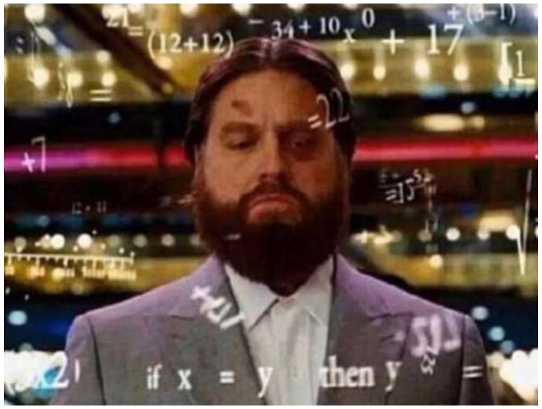
But we needs..

$$\frac{d\mathcal{L}}{dc_i} = \boxed{\frac{d\hat{C}}{d\sigma_i}}\frac{d\mathcal{L}}{d\hat{C}}$$

$$\hat{C}(\mathbf{r}) = \sum_{i=1}^{N} T_i(1 - \exp(-\sigma_i \delta_i))\mathbf{c}_i \,, \text{ where } T_i = \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right) , \qquad (3)$$

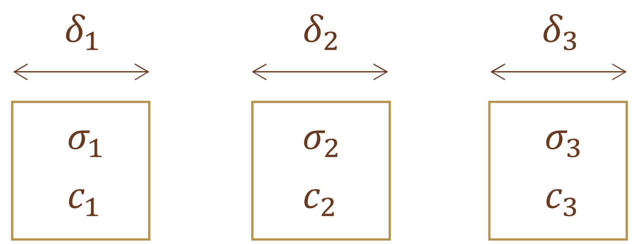
where $\delta_i = t_{i+1} - t_i$ is the distance between adjacent samples. This function for calculating $\hat{C}(\mathbf{r})$ from the set of $(\mathbf{c}_i, \sigma_i)$ values is trivially differentiable and reduces to traditional alpha compositing with alpha values $\alpha_i = 1 - \exp(-\sigma_i \delta_i)$.

NeRF paper – Eq. 3



Me when seeing that the paper explain it as trivial

Derivative $\frac{d\hat{C}}{dc_i}$



$$\hat{C} = T_1(1-\exp(-\sigma_1\delta_1))c_1 + T_2(1-\exp(-\sigma_2\delta_2))c_2 + T_3(1-\exp(-\sigma_3\delta_3))c_3$$

Fortunately, these derivatives are trivial indeed as follows:

$$\frac{d\hat{C}}{dc_1} = T_1(1-\exp(-\sigma_1\delta_1))$$

$$\frac{d\hat{C}}{dc_2} = T_2(1-\exp(-\sigma_2\delta_2))$$

$$\frac{d\hat{C}}{dc_3} = T_3(1-\exp(-\sigma_3\delta_3))$$

Derivative $\frac{d\hat{C}}{d\sigma_i}$

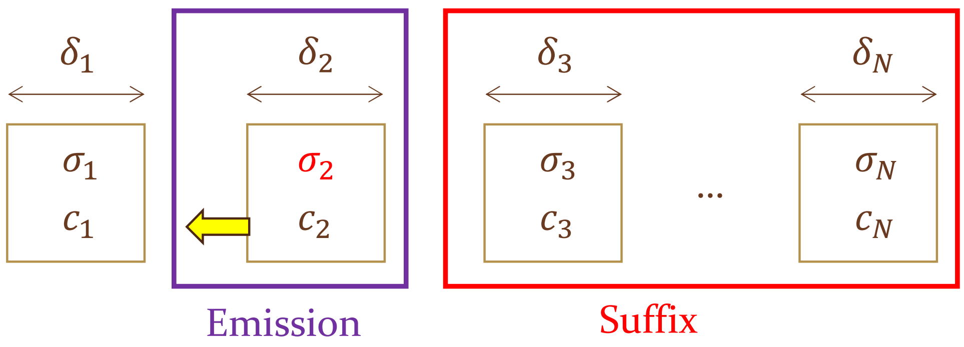
- The derivative consists of 3 major elements

Let's focus $\boldsymbol{\sigma_2}$

$T_3 = \exp(-\sigma_1\delta_1)\exp(-\sigma_2\delta_2)$

$T_i = \prod_{j=1}^{i-1} \exp(-\sigma_i\delta_i)$

$$\hat{C} = \underbrace{T_1(1-\exp(-\sigma_1\delta_1))c_1}_{\text{Constant.}} + \underbrace{T_2(1-\exp(-\sigma_2\delta_2))c_2}_{\text{Emission}} + \underbrace{T_3(1-\exp(-\sigma_3\delta_3))c_3 + \ldots\, T_N(1-\exp(-\sigma_N\delta_N))c_N}_{\text{Suffix}}$$



Derivative $\frac{d\hat{C}}{d\sigma_i}$ - Suffix

- Let's define suffix S_j

$$T_3 = \exp(-\sigma_1\delta_1)\exp(-\sigma_2\delta_2) \qquad T_i = \prod_{j=1}^{i-1} \exp(-\sigma_i\delta_i)$$

$$\hat{C} = T_1(1-\exp(-\sigma_1\delta_1))c_1 + T_2(1-\exp(-\sigma_2\delta_2))c_2 + \underbrace{T_3(1-\exp(-\sigma_3\delta_3))c_3 + \ldots T_N(1-\exp(-\sigma_N\delta_N))c_N}_{\text{Suffix } S_2}$$

$$S_2 = T_3(1-\exp(-\sigma_3\delta_3))c_3 + \ldots T_N(1-\exp(-\sigma_N\delta_N))c_N$$

$$S_j = \sum_{i=j+1}^{N} T_i(1-\exp(-\sigma_i\delta_i))c_i$$

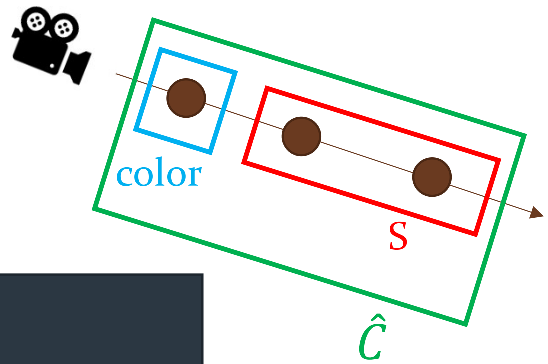
Key insight:

All terms in S_2 contains just an $\exp(-\sigma_2\delta_2)$ term inside T. Thus, the derivative is

$$S_2 = \exp(-\sigma_2\delta_2)(Constant)$$

$$\frac{dS_2}{d\sigma_2} = -\delta_2\exp(-\sigma_2\delta_2)(Constant) = -\delta_2 S_2$$

Derivative $\frac{d\hat{C}}{d\sigma_i}$ - Suffix



- Suffix S_j is cheaply available via $\hat{C}$

```
float3 color = make_float3( 0.0f, 0.0f, 0.0f );
for( int yi = eval_beg; yi < eval_end; yi++ )
{
    float dt = ...;
    float sigma = ...
    float3 c = ...
    float a = 1.0f - exp( -sigma * dt );
    float coefficient = T * a;
    color += coefficient * c;

    T *= ( 1.0f - a );

    float3 S = C_hat - color;

}
```

Derivative $\frac{d\hat{C}}{d\sigma_i}$ - Emission

- Almost trivial to get the derivative of the emission

$$\hat{C} = T_1(1-\exp(-\sigma_1\delta_1))c_1 + \underbrace{T_2(1-\exp(-\sigma_2\delta_2))c_2}_{\text{Emission}} + T_3(1-\exp(-\sigma_3\delta_3))c_3 + \ldots T_N(1-\exp(-\sigma_N\delta_N))c_N$$

$$\hat{C}_{emission} = T_2\big(1-\exp(-\sigma_2\delta_2)\big)c_2 = T_2c_2 - T_2c_2\exp(-\sigma_2\delta_2)$$

$$\frac{d\hat{C}_{emission}}{d\sigma_2} = T_2c_2\delta_2\exp(-\sigma_2\delta_2) = T_3c_2\delta_2$$

$$T_{i+1} = T_i\exp(-\sigma_i\delta_i)$$

Derivative of σ_i - Complete Form

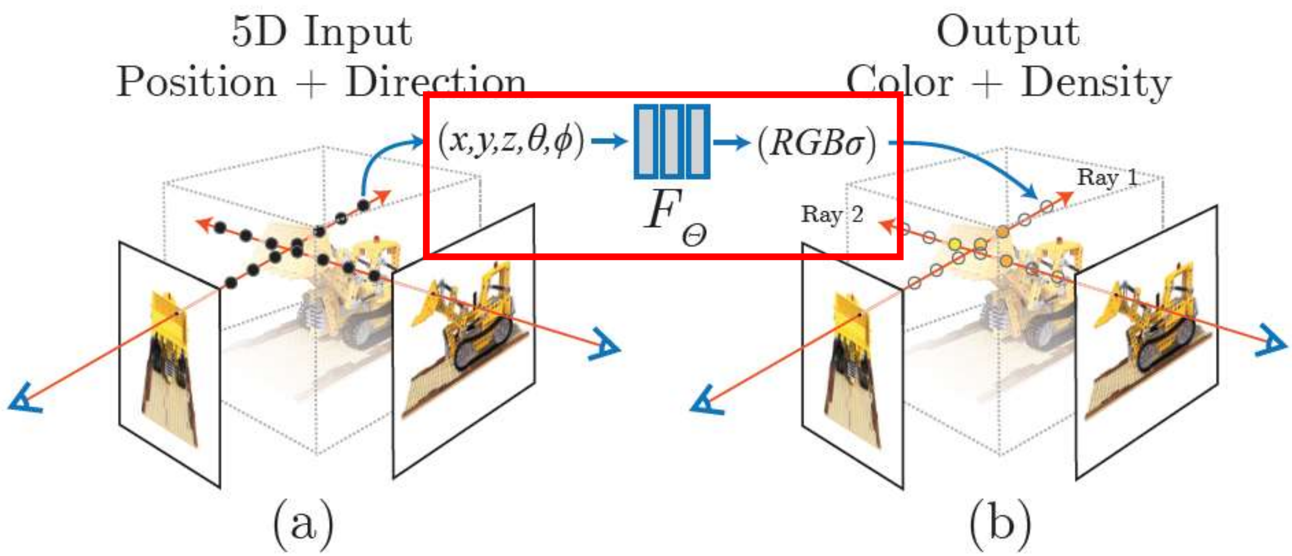
- Simple and very easy to evaluate ☺

$$\frac{d\hat{C}}{d\sigma_i} = \delta_i(T_{i+1}c_i - S_i) \Rightarrow$$ Propagate this to the neural network

Intuitive explanation

Operation	Result
Increase σ	Make it more opaque then suppress the subsequent color contributions.
Decrease σ	Make it more transparent then take more contributions from the subsequent colors.

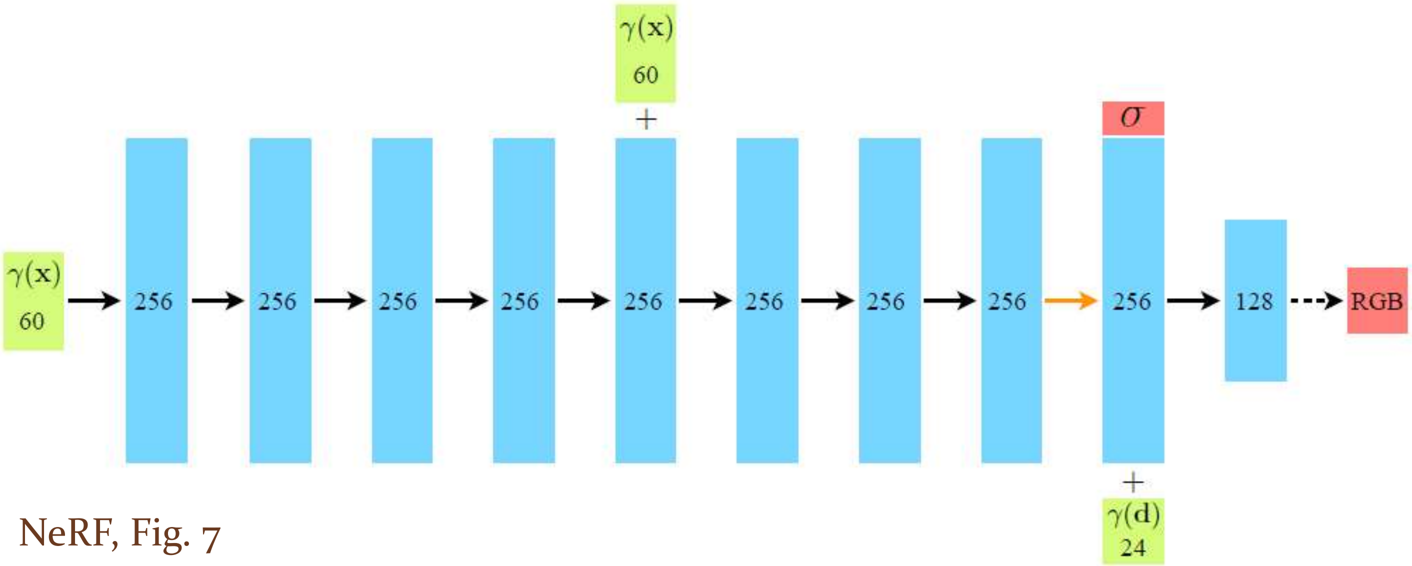
Neural network



NeRF paper - Fig. 2

Neural network – the original proposal

- 256 dimension, 9 layers + 128 dim layer
- Expensive to evaluate and train



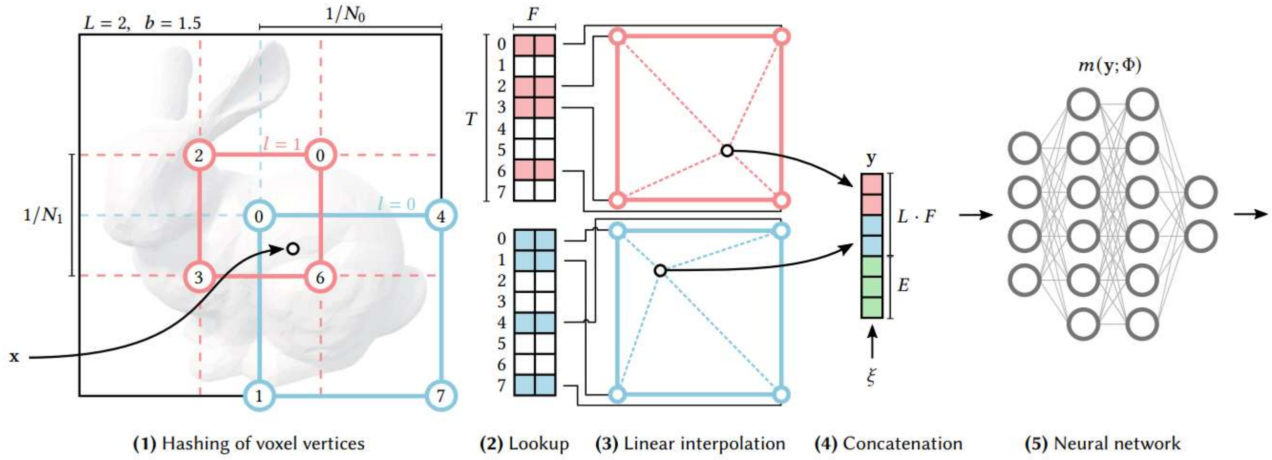
NeRF, Fig. 7

Neural network – Instant Neural Graphics Primitives with a Multiresolution Hash Encoding

- A novel input preprocess technique for neural network
- Reduce network size without sacrificing quality
  - Produce even better quality
- Simple to implement

The idea

- Define grids with different resolutions for the input space
- Each grid corner is assigned to elements on the hash table
- Feed the linearly interpolated value to the neural network



Instant NGP, Fig. 3

Random thoughts

Then, the role of the Neural Network is just a tiny adjustment?

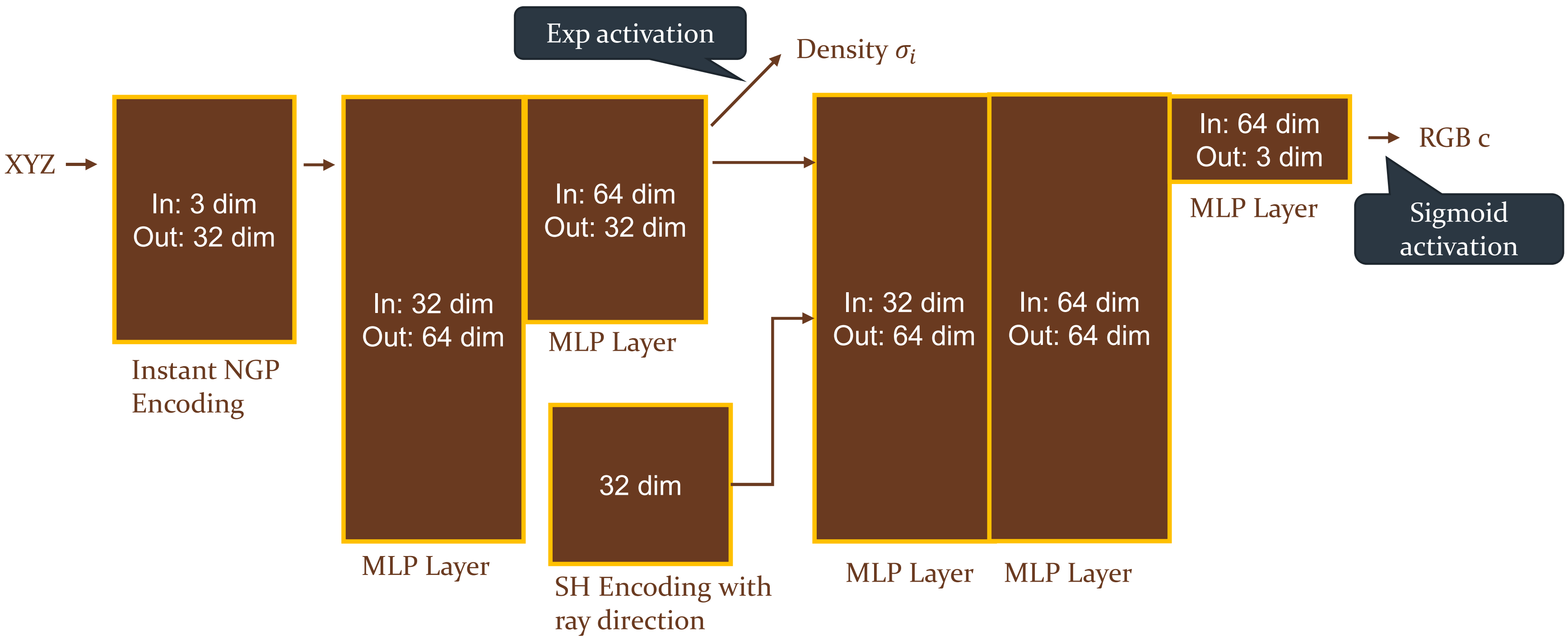
Hash Grid

This can be replaced by something great idea?

Neural Network

The network architecture

- https://github.com/NVlabs/instant-ngp



GPU implementation

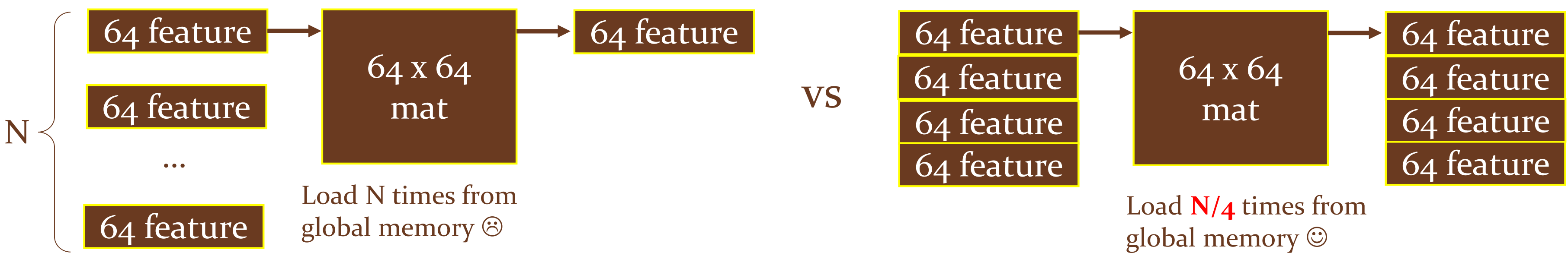
- Almost of the calculation can be done with matrix multiplications
- A smaller network architecture such as for NeRF is suitable for the GPU

Thread-group matrix multiplications

- Explained at “Real-time Neural Radiance Caching for Path Tracing”

But why?

- Feature vectors and weighting matrix on regs such as 64 wide consume too many vector registers ☹
- Batch multiplications can reduce the bandwidth for weighting matrix ☺
  - Weighting matrix can be large



Thread-group matrix multiplications (Software)

Goal:

$$\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix}\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} =$$

$$\begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & x_{11}w_{13} + x_{12}w_{23} \\ x_{21}w_{11} + x_{22}w_{21} & x_{21}w_{12} + x_{22}w_{22} & x_{21}w_{13} + x_{22}w_{23} \\ x_{31}w_{11} + x_{32}w_{21} & x_{31}w_{12} + x_{32}w_{22} & x_{31}w_{13} + x_{32}w_{23} \end{pmatrix}$$

Note:
Feature Vector is on shared memory
Weighting Matrix is on global memory

Input → Layer 1 Mat → Layer 2 Mat → Output

1.The output matrix is separately assigned to threads

Thread 0	Thread 1	Thread 2
0	0	0
0	0	0
0	0	0

Thread-group matrix multiplications (Software)

Goal:

$$\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} =$$

$$\begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & x_{11}w_{13} + x_{12}w_{23} \\ x_{21}w_{11} + x_{22}w_{21} & x_{21}w_{12} + x_{22}w_{22} & x_{21}w_{13} + x_{22}w_{23} \\ x_{31}w_{11} + x_{32}w_{21} & x_{31}w_{12} + x_{32}w_{22} & x_{31}w_{13} + x_{32}w_{23} \end{pmatrix}$$

2. Load a weight from the global memory for each thread

Thread 0	Thread 1	Thread 2
0 0 0	0 0 0	0 0 0

w_{11} $\quad$ w_{12} $\quad$ w_{12}

Thread-group matrix multiplications (Software)

Goal:

$$\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} =$$

$$\begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & x_{11}w_{13} + x_{12}w_{23} \\ x_{21}w_{11} + x_{22}w_{21} & x_{21}w_{12} + x_{22}w_{22} & x_{21}w_{13} + x_{22}w_{23} \\ x_{31}w_{11} + x_{32}w_{21} & x_{31}w_{12} + x_{32}w_{22} & x_{31}w_{13} + x_{32}w_{23} \end{pmatrix}$$

3. Load a column of the feature vectors from shared memory, and do multiply-add operations

Thread 0	Thread 1	Thread 2
$x_{11}w_{11}$ $x_{21}w_{11}$ $x_{31}w_{11}$	$x_{11}w_{12}$ $x_{21}w_{12}$ $x_{31}w_{12}$	$x_{11}w_{13}$ $x_{21}w_{13}$ $x_{31}w_{13}$
w_{11}	w_{12}	w_{12}

Thread-group matrix multiplications (Software)

Goal:

$$\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ \boxed{w_{21} \quad w_{22} \quad w_{23}} \end{pmatrix} =$$

$$\begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & x_{11}w_{13} + x_{12}w_{23} \\ x_{21}w_{11} + x_{22}w_{21} & x_{21}w_{12} + x_{22}w_{22} & x_{21}w_{13} + x_{22}w_{23} \\ x_{31}w_{11} + x_{32}w_{21} & x_{31}w_{12} + x_{32}w_{22} & x_{31}w_{13} + x_{32}w_{23} \end{pmatrix}$$

4. Load a weight from the global memory for each thread

Thread 0	Thread 1	Thread 2
$x_{11}w_{11}$ $x_{21}w_{11}$ $x_{31}w_{11}$	$x_{11}w_{12}$ $x_{21}w_{12}$ $x_{31}w_{12}$	$x_{11}w_{13}$ $x_{21}w_{13}$ $x_{31}w_{13}$

w_{21} $\qquad$ w_{22} $\qquad$ w_{22}

Thread-group matrix multiplications (Software)

Goal:

$$\begin{pmatrix} x_{11} & \boxed{x_{12}} \\ x_{21} & \boxed{x_{22}} \\ x_{31} & \boxed{x_{32}} \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} =$$

$$\begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & x_{11}w_{13} + x_{12}w_{23} \\ x_{21}w_{11} + x_{22}w_{21} & x_{21}w_{12} + x_{22}w_{22} & x_{21}w_{13} + x_{22}w_{23} \\ x_{31}w_{11} + x_{32}w_{21} & x_{31}w_{12} + x_{32}w_{22} & x_{31}w_{13} + x_{32}w_{23} \end{pmatrix}$$

5. Load a column of the feature vectors from shared memory, and do multiply-add operations

Thread 0	Thread 1	Thread 2
$x_{11}w_{11} + x_{12}w_{21}$	$x_{11}w_{12} + x_{12}w_{22}$	$x_{11}w_{13} + x_{12}w_{23}$
$x_{21}w_{11} + x_{22}w_{21}$	$x_{21}w_{12} + x_{22}w_{22}$	$x_{21}w_{13} + x_{22}w_{23}$
$x_{31}w_{11} + x_{32}w_{21}$	$x_{31}w_{12} + x_{32}w_{22}$	$x_{31}w_{13} + x_{32}w_{23}$

w_{21} $\qquad$ w_{22} $\qquad$ w_{22}

Thread-group matrix multiplications (Software)

Goal: $\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{pmatrix} \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} =$

The weighting matrix is loaded from the global memory exactly once ☺

$$\begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & x_{11}w_{13} + x_{12}w_{23} \\ x_{21}w_{11} + x_{22}w_{21} & x_{21}w_{12} + x_{22}w_{22} & x_{21}w_{13} + x_{22}w_{23} \\ x_{31}w_{11} + x_{32}w_{21} & x_{31}w_{12} + x_{32}w_{22} & x_{31}w_{13} + x_{32}w_{23} \end{pmatrix}$$

Vector register allocation is modest ☺

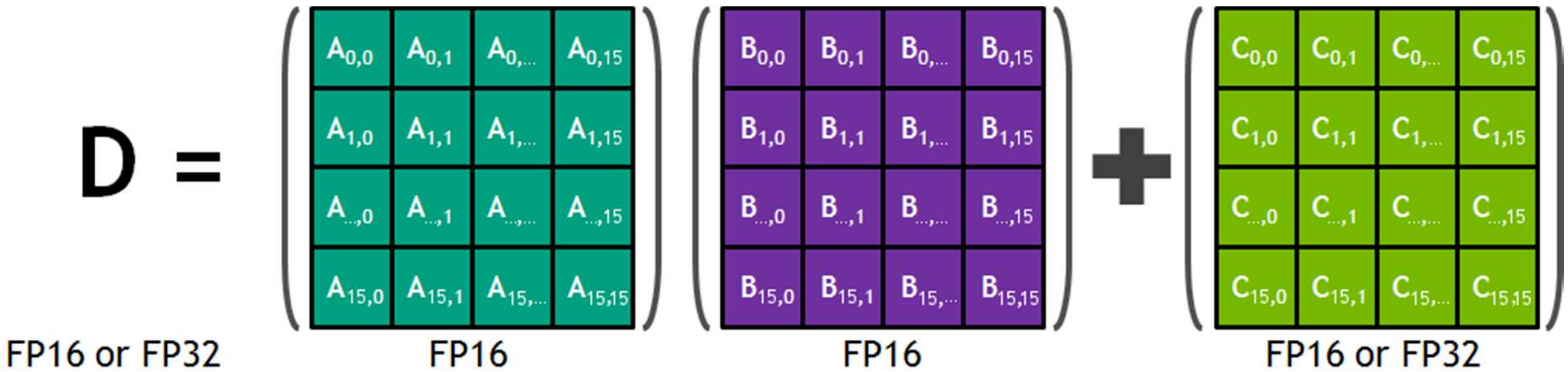
Done!

Roughly 4x ~ 5x faster than the naive “an item per thread” approach.

Thread 0	Thread 1	Thread 2
$x_{11}w_{11} + x_{12}w_{21}$	$x_{11}w_{12} + x_{12}w_{22}$	$x_{11}w_{13} + x_{12}w_{23}$
$x_{21}w_{11} + x_{22}w_{21}$	$x_{21}w_{12} + x_{22}w_{22}$	$x_{21}w_{13} + x_{22}w_{23}$
$x_{31}w_{11} + x_{32}w_{21}$	$x_{31}w_{12} + x_{32}w_{22}$	$x_{31}w_{13} + x_{32}w_{23}$

Thread-group matrix multiplications (Hardware)

- WMMA on NV platform
- Fixed size matrix multiplications for each warp e.g. 16x16 matrix



https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

Thread-group matrix multiplications (Hardware)

The idea is simple:

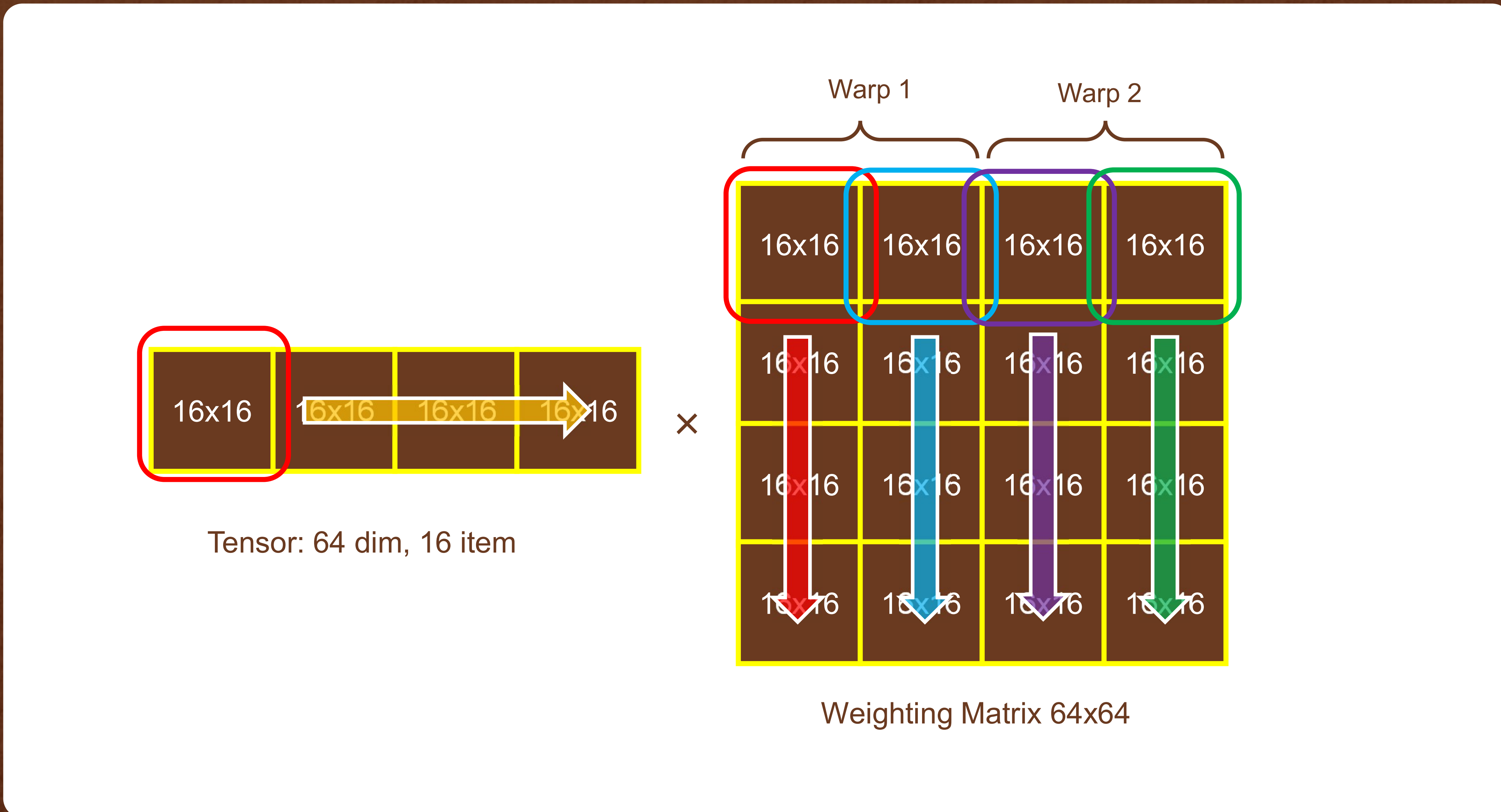
- Separate a matrix multiplication into smaller matrix multiplications

Step 1

$$\begin{pmatrix} \boxed{\begin{matrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{matrix}} & \begin{matrix} x_{13} & x_{14} \\ x_{23} & x_{24} \end{matrix} \end{pmatrix} \begin{pmatrix} \boxed{\begin{matrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{matrix}} & \begin{matrix} w_{13} & w_{14} \\ w_{23} & w_{24} \end{matrix} \\ \begin{matrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{matrix} & \begin{matrix} w_{33} & w_{34} \\ w_{43} & w_{44} \end{matrix} \end{pmatrix} = \begin{pmatrix} x_{11}w_{11} + x_{12}w_{21} & x_{11}w_{12} + x_{12}w_{22} & .. & .. \\ x_{22}w_{11} + x_{22}w_{21} & x_{21}w_{12} + x_{22}w_{22} & .. & .. \end{pmatrix}$$

Step 2

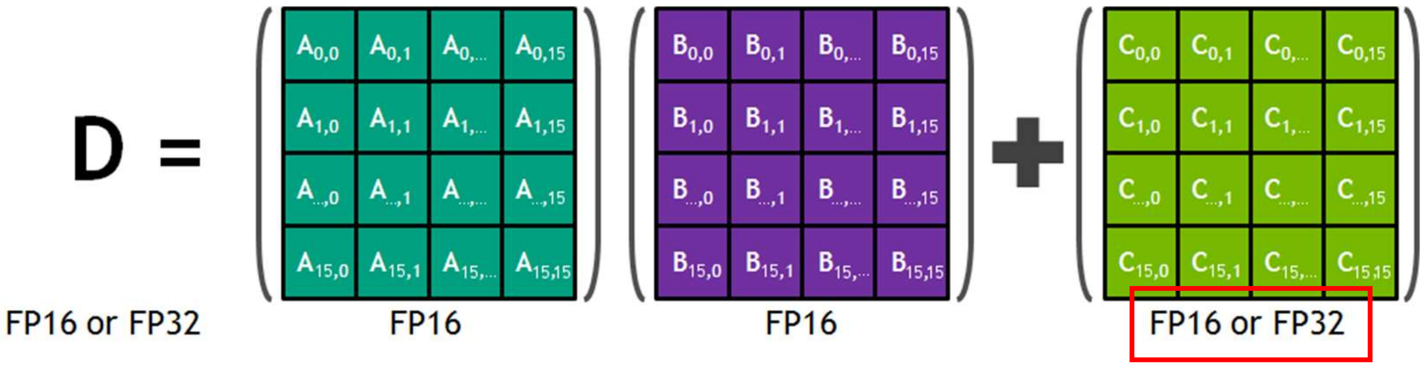
$$\begin{pmatrix} \begin{matrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{matrix} & \boxed{\begin{matrix} x_{13} & x_{14} \\ x_{23} & x_{24} \end{matrix}} \end{pmatrix} \begin{pmatrix} \begin{matrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{matrix} & \begin{matrix} w_{13} & w_{14} \\ w_{23} & w_{24} \end{matrix} \\ \boxed{\begin{matrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{matrix}} & \begin{matrix} w_{33} & w_{34} \\ w_{43} & w_{44} \end{matrix} \end{pmatrix} = \begin{pmatrix} \ldots + x_{13}w_{31} + x_{14}w_{41} & \ldots + x_{13}w_{12} + x_{14}w_{22} & .. & .. \\ \ldots + x_{23}w_{31} + x_{24}w_{41} & \ldots + x_{23}w_{32} + x_{24}w_{42} & .. & .. \end{pmatrix}$$



Warp 1
Warp 2
16x16
16x16
16x16
16x16
16x16
16x16
16x16
16x16
×
Tensor: 64 dim, 16 item
Weighting Matrix 64x64

Remark

- Roughly 1.3x faster than the software on my nerf implementation
  - Not significant, because hash grid encoding is also bottleneck
  - May need more investigations
- The accumulation can be done with FP32
  - Depends on applications, but I couldn't find any numerical issue



Summary

- The volume formulation on NeRF is simple and powerful
  - The implementation is not rocket science
- GPU implementation makes sense
  - Thread-group matrix multiplications achieve great performance
    - Even without dedicated HW
  - WMMA can improve more
    - You have any ideas? Please let me know

Questions?

References

- Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi and Ren Ng, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis"
- Thomas Müller, Fabrice Rousselle, Jan Novák, and Alex Keller, "Real-time Neural Radiance Caching for Path Tracing"
- Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding"
- "Instant NGP" repository : https://github.com/NVlabs/instant-ngp
- Takahiro Harada and Aaryaman Vasishta, "Introducing Orochi" https://gpuopen.com/learn/introducing-orochi/
- "Orochi" repository : https://github.com/GPUOpen-LibrariesAndSDKs/Orochi
- Jeremy Appleyard and Scott Yokim, "Programming Tensor Cores in CUDA 9" : https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/