
# INTRODUCTION TO NEURAL SHADING

SIGGRAPH 2025
Vancouver+ 10-14 August

# INTRODUCTION

# WHY NEURAL SHADING?



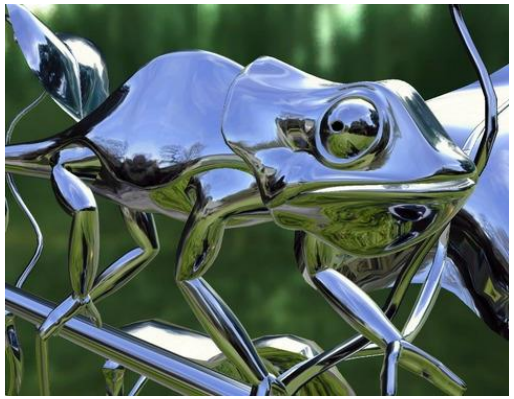
- Why we’re all here: Rendering the best possible image

# WHY NEURAL SHADING?



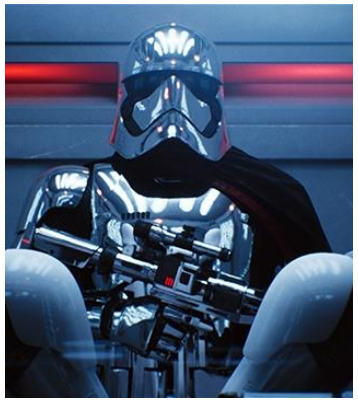
- Why we're all here: Rendering the best possible image in ~~16ms~~ 33ms
- 40+ years of hard work on rendering, content and hardware

[Cook, 1984]



2000

2007



2019

# WHY NEURAL SHADING?

>10,000X
Faster in 9 Years

100X GPU and Drivers

10X DLSS

30X Rendering Algorithms

1X

Pascal
2016

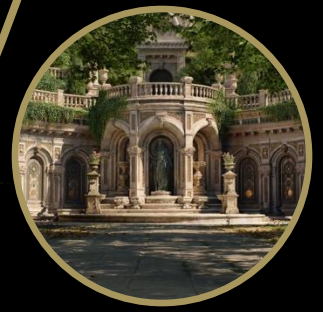
Turing
2018

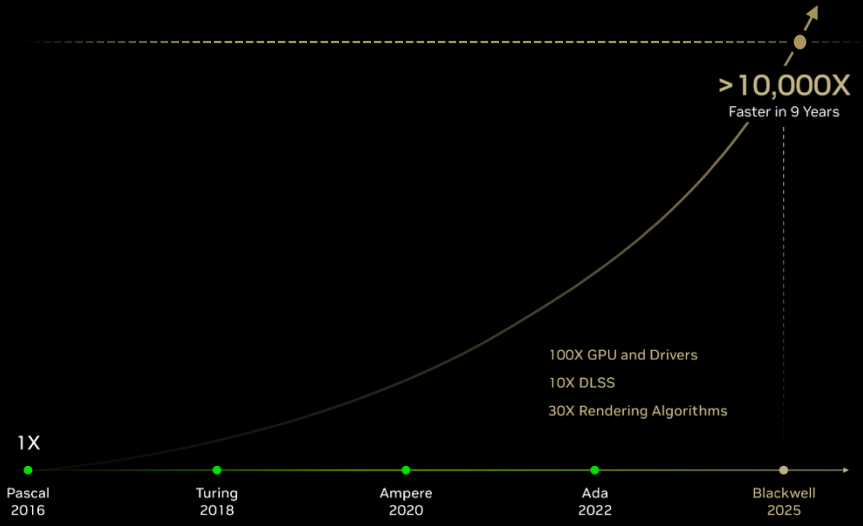
Ampere
2020

Ada
2022

Blackwell
2025

# WHY NEURAL SHADING?






- Some of the fuel for this growth was Moore’s law
- Moore’s law has ended
- We need new tools!

# WHAT IS NEURAL SHADING?



- Usually: Involves Neural Networks
- Anything that's trainable

Neural Shader

# WHAT IS NEURAL SHADING?



- Usually: Involves Neural Networks
- Anything that's trainable

Neural Shader

- Runs in the graphics pipeline
- Part of your normal shader code

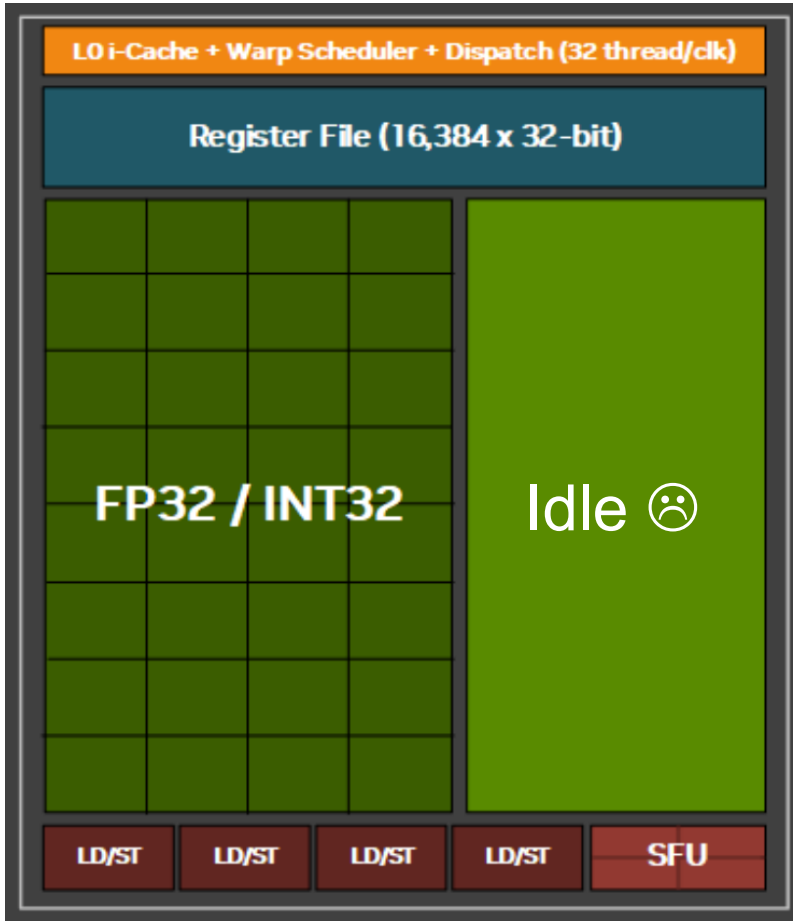
# HOW DO NEURAL SHADERS HELP?



- Consumer GPUs ship with neural network accelerators
- While you render, these sit at 0% utilization

- These are untapped FLOPS!
- …and very efficient FLOPS

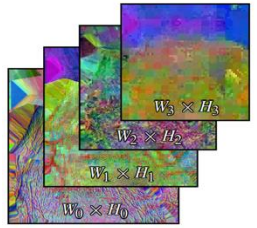
- Accessible in the graphics pipeline with Cooperative Vectors



# HOW DO NEURAL SHADERS HELP?



[Fujieda and Harada, 2024]



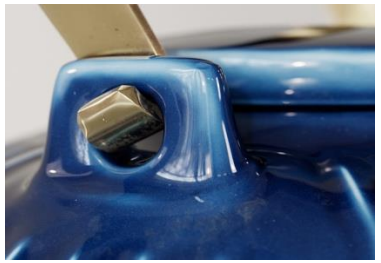

[Belcour and Benyoub, 2025]

[Vaidyanathan et al., 2023]

## Compression

[Kuznetsov et al., 2021]

[Mullia et al., 2024]



[Zeltner et al., 2024]

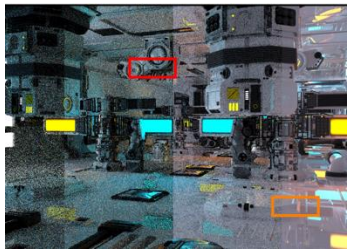
## Materials

[Mildenhall et al., 2020]

[Müller et al., 2022]

[Kerbl et al., 2023]

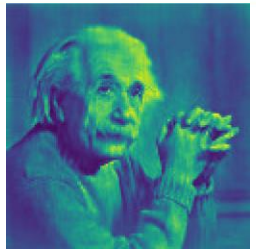
## Geometry



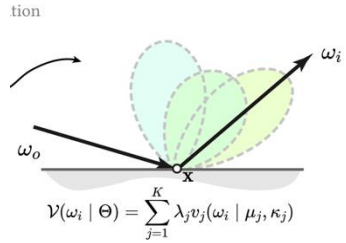
[Müller et al., 2021]

[Dereviannykh et al., 2024]

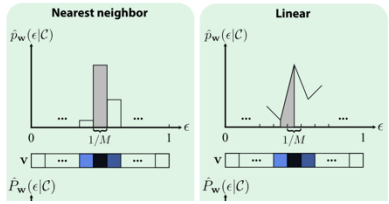
## Caching



[Müller et al., 2019]



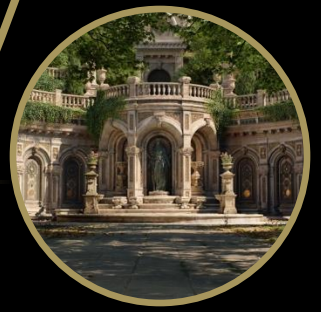

[Dong et al., 2023]

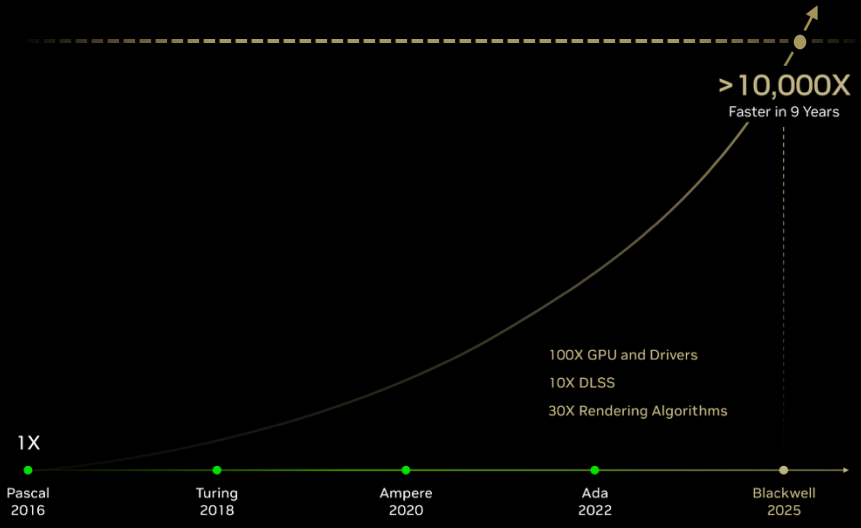



[Figueiredo et al., 2025]

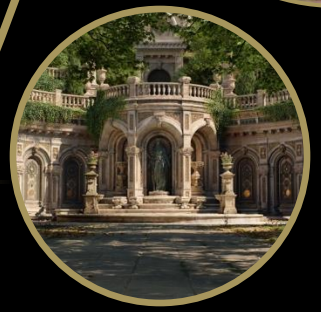
## Guiding

# WHY NEURAL SHADING?

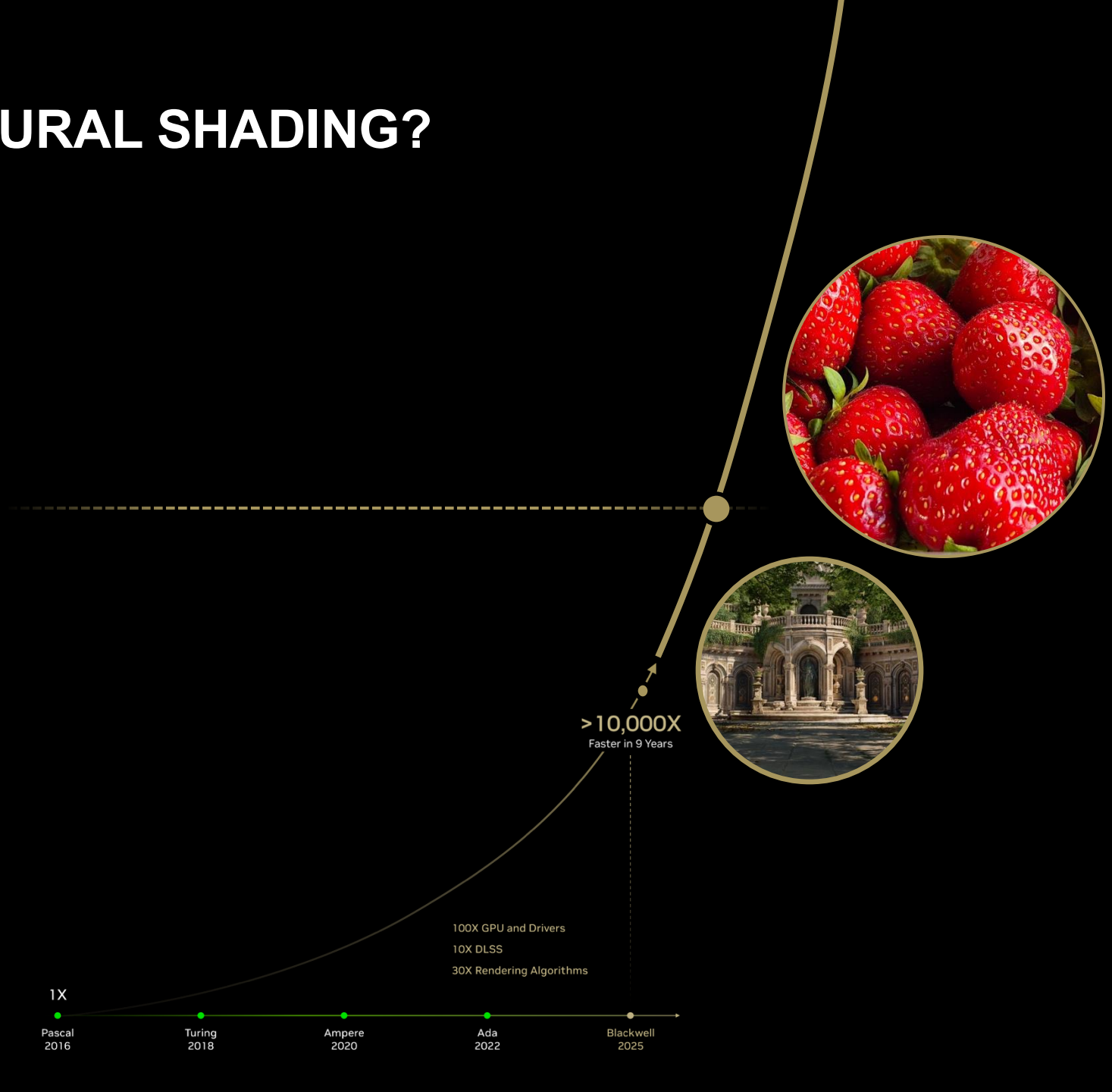




# WHY NEURAL SHADING?



# WHY NEURAL SHADING?



# SUMMARY



Neural Shaders are a powerful new tool in your tool box:

- Renderers that are trainable can solve problems that were hard before
- Neural Networks can run very efficiently
- This is a practical tool worth checking out

# SUMMARY



But: Neural Shaders are not magic

- Things you already do well won’t magically get better
- We need new tools, new programming models and new algorithms
- Leveraging the full power small neural nets requires new skills and clever engineering

# TODAY'S COURSE



How to train and deploy neural shaders, from the ground up:

- Writing your first trainable shader
- …and first neural network
- How to debug
- Hardware acceleration with Cooperative Vectors
- Neural Texture
- Neural Materials

- No pre-existing knowledge needed

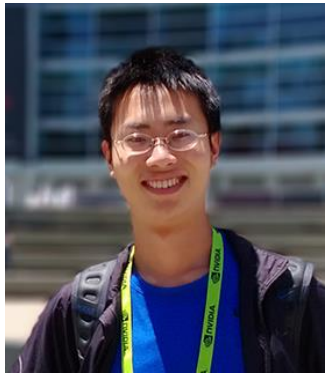
SIGGRAPH 2025
Vancouver+ 10-14 August

# MECHANICS OF TODAY

# SPEAKERS FOR TODAY

SIGGRAPH 2025
Vancouver+ 10-14 August


**Benedikt Bitterli**
bbitterli@nvidia.com



**Yong He**
yhe@nvidia.com

**Chris Cummings**
ccummings@nvidia.com

**Alexey Panteleev**
alpanteleev@nvidia.com

**Alexey Bekin**
abekin@nvidia.com

**Brandon Mills**
bmills@nvidia.com

# TODAY'S AGENDA



9:00am – 10:45am

- Fundamentals
- Core Concepts
- Automatic Differentiation

10:45am – 12:15pm

- Hardware Acceleration
- Performance Tips
- Deployment
- Neural Textures
- Neural Materials

**Course Material:**

https://shader-slang.org/landing/siggraph-25

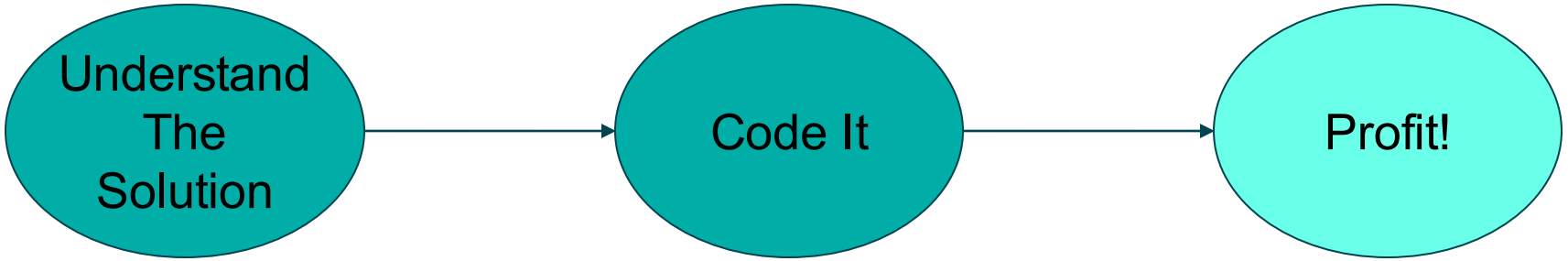
SIGGRAPH 2025
Vancouver+ 10-14 August

# FUNDAMENTALS

# WHAT IF WE DON'T KNOW THE ANSWER?



## Classical Engineering



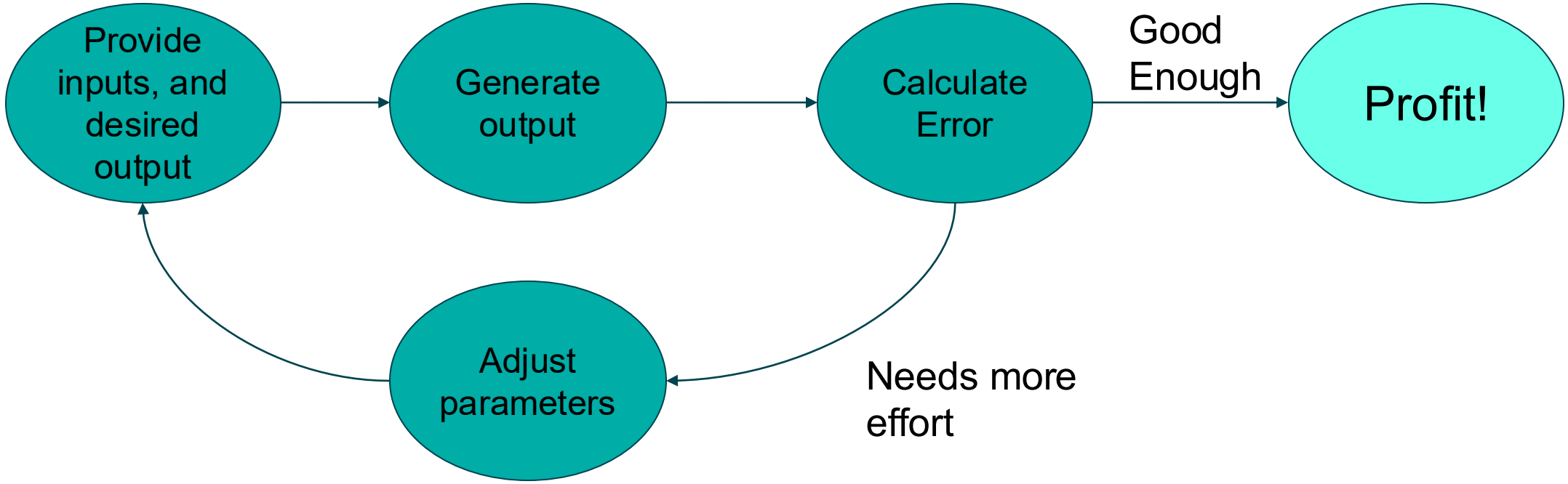

## But what if…

- We don't know the solution
- Or there isn't one!
- Or there is, but it's **way** too expensive to compute

# LEARNING AN APPROXIMATE SOLUTION.



## Optimizer…



# TOOLS FOR TODAY




- Powerful / flexible shading language
- Write once / run anywhere
- Generics
- Supports ‘auto-diff’ (does calculus for us)

**SlangPy**

- Python/C++ interface to Slang
- Full featured graphics api
- Cross platform
- Functional api to directly call Slang functions from Python

# CAN WE LEARN BETTER MIP MAPS?



- Mipmaps reduce aliasing **and** improve coherency for distant surfaces
- Color maps downsample well using simple box filter
- Normal maps do not

# CAN WE LEARN BETTER MIP MAPS?

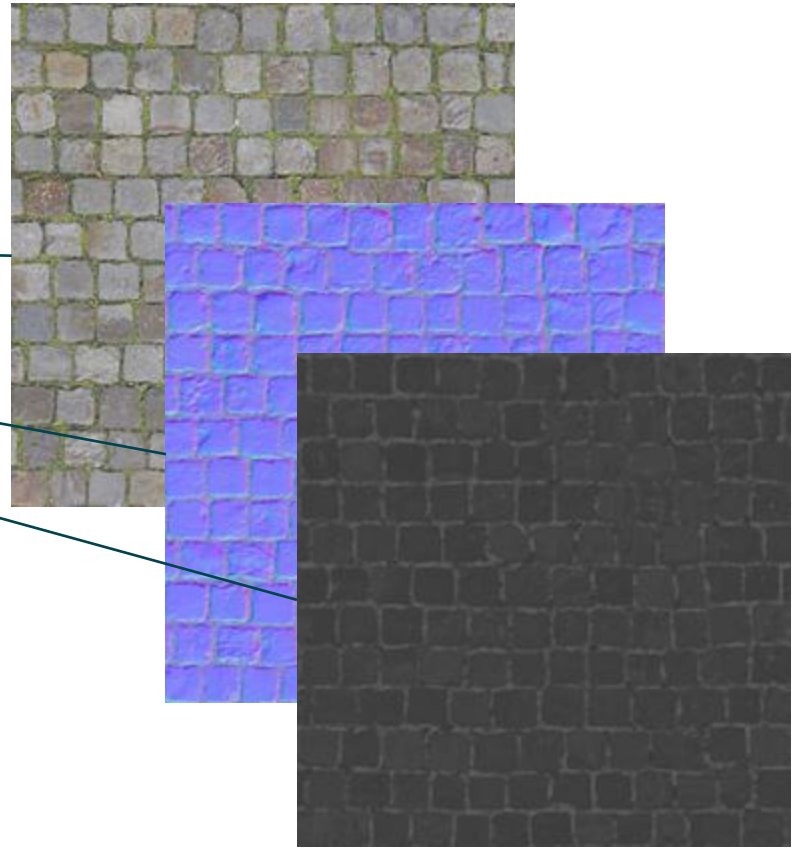


1. Render a beautiful PBR material
2. Generate + render low res with **Mip Maps**
3. Generate + render reference with **Supersampling**
4. Calculate the loss

# BASIC SLANG PROGRAM



```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;
    Tensor<float3, 2> normal;
    Tensor<float, 2> roughness;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }
    float3 get_normal(int2 pixel)
    {
        return normalize(normal.getv(pixel));
    }
    float get_roughness(int2 pixel)
    {
        return roughness.getv(pixel);
    }
};
```



# BASIC SLANG PROGRAM



```
float3 render(int2 pixel, MaterialParameters material, float3 light_dir, float3 view_dir)
{
    // Bright white light
    float light_intensity = 5.0;

    // Sample very shiny BRDF (it rained today!)
    float3 brdf_sample = sample_brdf(
              material.get_albedo(pixel),  // albedo color
              normalize(light_dir),        // light direction
              normalize(view_dir),         // view direction
              material.get_normal(pixel),  // normal map sample
              0.05,                        // roughness
              0.0,                         // metallic (no metal)
              1.0                          // specular
              );

    // Combine light with BRDF sample to get pixel colour
    return brdf_sample * light_intensity;
}
```

# BASIC PYTHON PROGRAM



```python
# Create the app and load the slang module.
app = App(width=2048, height=2048, title="Mipmap Example")
module = spy.Module.load_from_file(app.device, "nsc_basicprogram.slang")
```

```python
# Load some materials.
albedo_map = spy.Tensor.load_from_image(app.device, "PavingStones070_2K.diffuse.jpg", linearize=True)
normal_map = spy.Tensor.load_from_image(app.device, "PavingStones070_2K.normal.jpg", scale=2, offset=-1)
roughness_map = spy.Tensor.load_from_image(app.device, "PavingStones070_2K.roughness.jpg", grayscale=True)
```

```python
while app.process_events():
    # Allocate a tensor for output + call the render function
    output = spy.Tensor.empty_like(albedo_map)
    module.render(pixel = spy.call_id(),
                  material = {
                        "albedo": albedo_map,
                        "normal": normal_map,
                        "roughness": roughness_map,
                  },
                  light_dir = spy.math.normalize(spy.float3(0.2, 0.2, 1.0)),
                  view_dir = spy.float3(0, 0, 1),
                  _result = output)

    # Blit tensor to screen.
    app.blit(output)
    app.present()
```

# RESULT!



## Conveniently Shiny Cobblestones!

# DOWNSAMPLING



```python
def downsample(source: spy.Tensor, steps: int) -> spy.Tensor:
    for i in range(steps):
        dest = spy.Tensor.empty(
                    device=app.device,
                    shape=(source.shape[0] // 2, source.shape[1] // 2),
                    dtype=source.dtype)
        module.downsample(spy.call_id(), source, _result=dest)
        source = dest
    return source
```

```
float3 downsample(
      int2 pixel,
      Tensor<float3, 2> source)
{
    float3 res = 0;
    res += source.getv(pixel * 2 + int2(0, 0));
    res += source.getv(pixel * 2 + int2(1, 0));
    res += source.getv(pixel * 2 + int2(0, 1));
    res += source.getv(pixel * 2 + int2(1, 1));
    return res * 0.25;
}
```

# DOWNSAMPLED INPUTS



```
# … downsampled output generation/render here

# Quarter res rendered output BRDF from half res inputs.
lr_output = spy.Tensor.empty_like(output)
module.render(pixel = spy.call_id(),
              material = {
                    "albedo":    downsample(albedo_map, 2),
                    "normal":    downsample(normal_map, 2),
                    "roughness": downsample(roughness_map, 2),
              },
              light_dir = spy.math.normalize(spy.float3(0.2, 0.2, 1.0)),
              view_dir = spy.float3(0, 0, 1),
              _result = lr_output)

# … blit the ‘lr_output’ to screen here
```

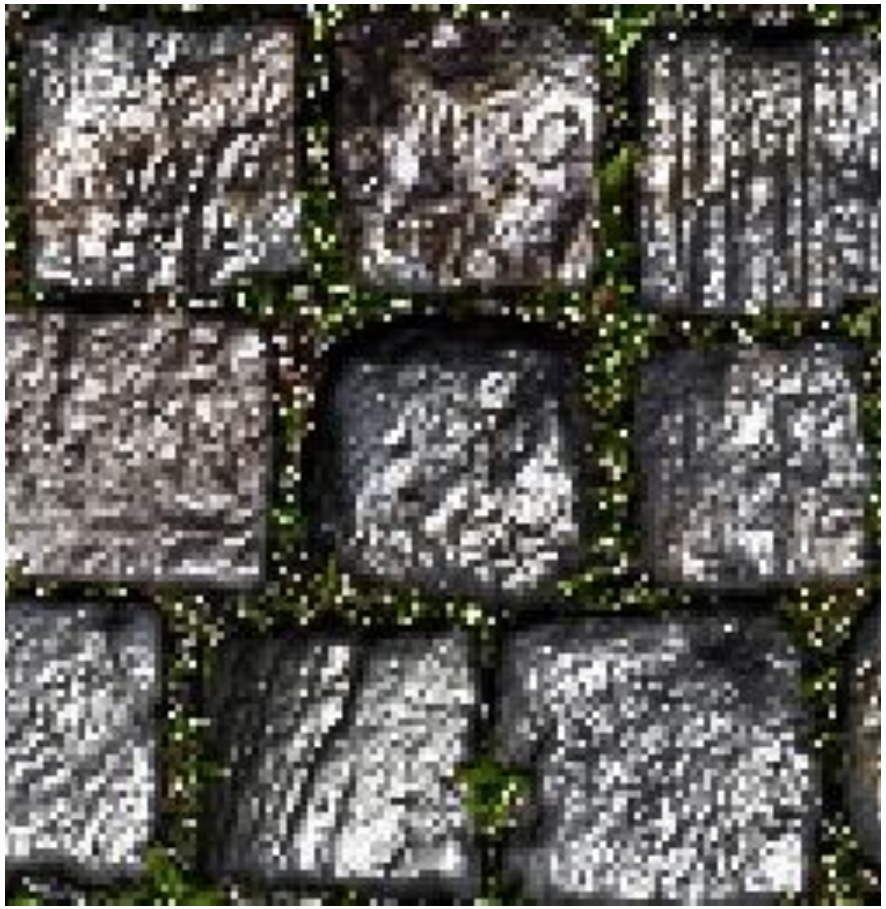
# RESULT!



**Downscaled inputs**

- 2x downsample (Mip level 2)
- 1 sample per map
- PBR function run once per pixel
- Optimal but noisy

Poor normal map downsampling gives nasty surface artifacts.



# DOWNSAMPLED OUTPUT



```
while app.process_events():
    # Allocate a tensor for output + call the render function
    output = spy.Tensor.empty_like(albedo_map)
    module.render(pixel = spy.call_id(),
                  material = {
                        "albedo": albedo_map,
                        "normal": normal_map,
                        "roughness": roughness_map
                  },
                  light_dir = spy.math.normalize(spy.float3(0.2, 0.2, 1.0)),
                  view_dir = spy.float3(0, 0, 1),
                  _result = output)

    # Downsample the output tensor to quarter res.
    output = downsample(output, 2)

    # Blit tensor to screen.
    app.blit(output, size=spy.int2(2048,2048))
    app.present()
```

# RESULT!



**Downscaled output**

- 2x downsample (Mip level 2)
- 16 high res albedo, normal + roughness pixels sampled
- Full PBR function run 16 times
- Represents an **ideal** output

If our input mip maps were *perfect*, this is what we'd get

# SIDE BY SIDE



**The Ideal**
Render at full res then downsample

**The Reality**
Downsample inputs and then render

# CALCULATING THE LOSS



```
// Render the difference between the rendered pixel
a reference pixel.
float3 loss(
      int2 pixel,
      float3 reference,
      MaterialParameters material,
      float3 light_dir,
      float3 view_dir)
{
    float3 color = render(pixel, material,
                          light_dir, view_dir);
    float3 error = color - reference;
    return error * error; // Squared error
}
```

```
# Loss between downsampled full res output (the
# reference), and result from quarter res inputs.
loss_output = spy.Tensor.empty_like(output)
module.loss(pixel = spy.call_id(),
            material = {
                  "albedo": downsample(albedo_map,2),
               "normal": downsample(normal_map,2),
               "roughness": downsample(roughness_map,2),
            },
            reference = output,
            light_dir = spy.math.normalize(
                      spy.float3(0.2, 0.2, 1.0)),
            view_dir = spy.float3(0, 0, 1),
            _result = loss_output)
```

# CALCULATING THE LOSS



**Reference**

**Loss**

**Render**

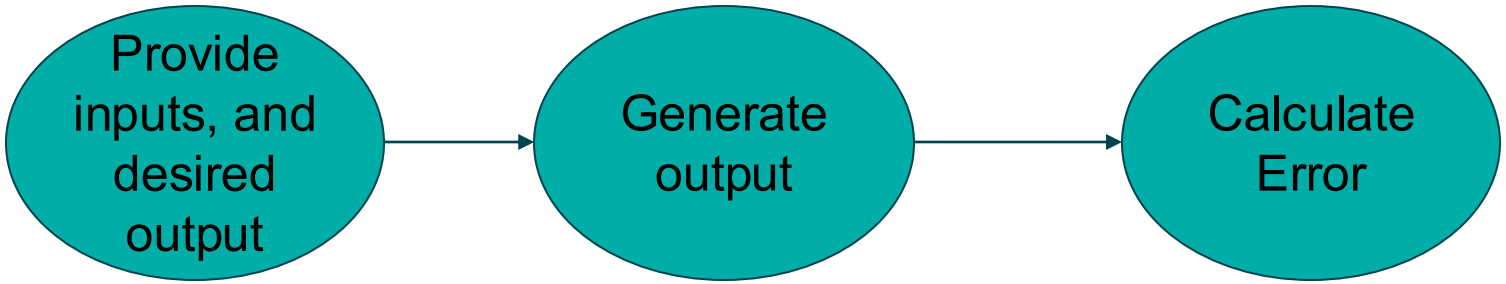
Make mipmap better == Make the loss smaller

SIGGRAPH 2025
Vancouver+ 10-14 August

# CORE CONCEPTS

# SUMMARY: THE “INFERENCE” PHASE



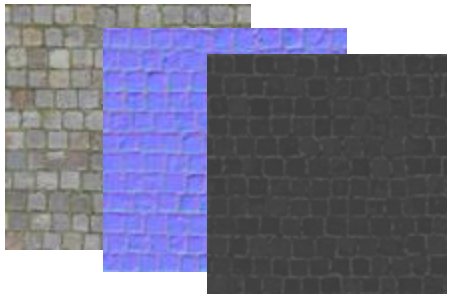


# SUMMARY: THE “FORWARD” PHASE



**Trainable Parameters**

```
MaterialParameters material
```



**Generate output**

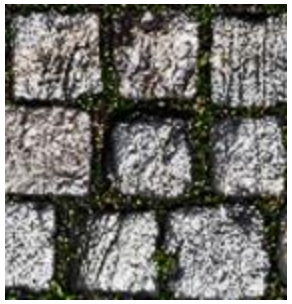
```
float3 render( /* ... */ )
```

**Compute error**

```
float3 loss( /* ... */ )
```

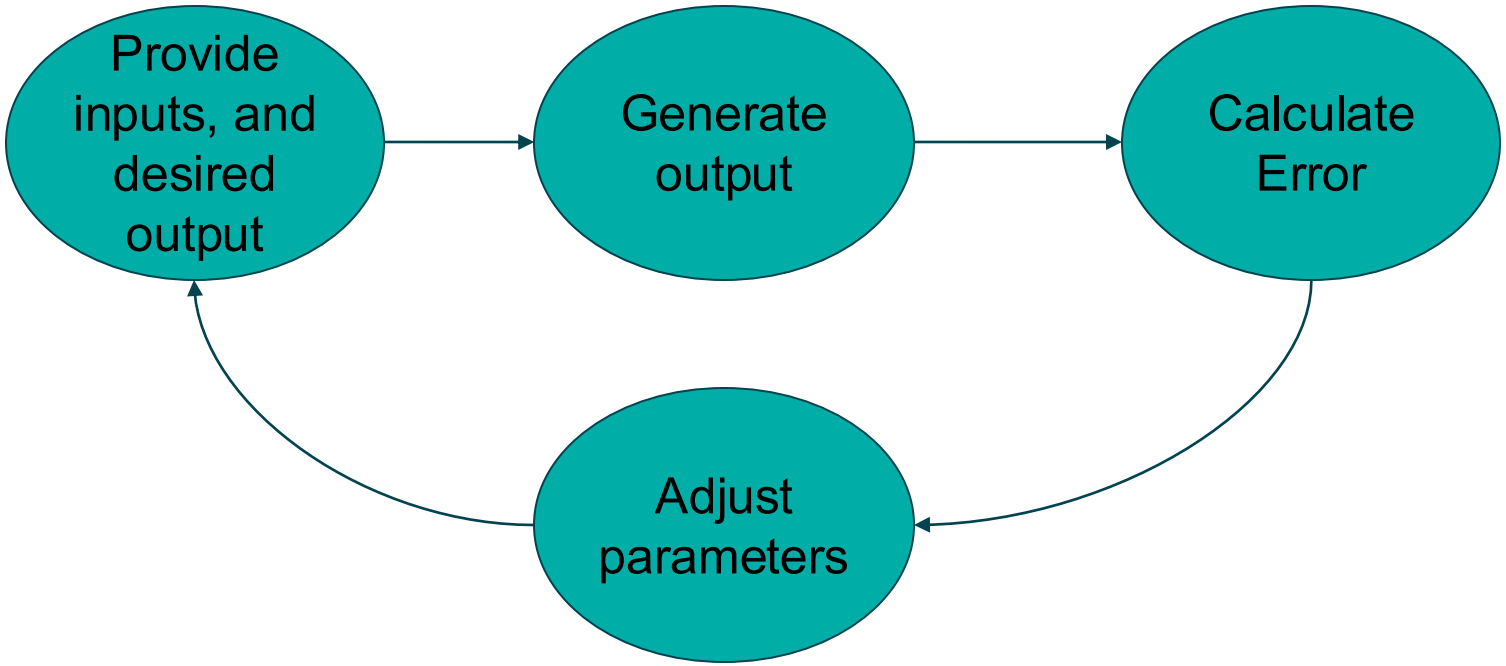
**Desired output**

```
float3 reference
```



# CLOSING THE LOOP: OPTIMIZATION



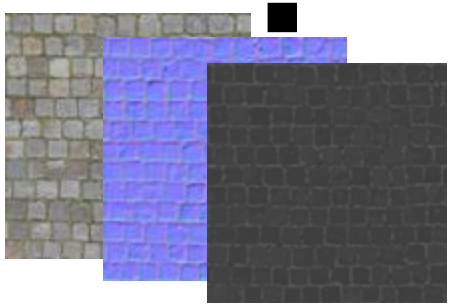


# THE “BACKWARD” PHASE



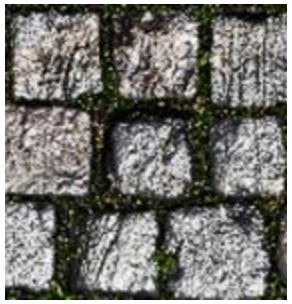
- **Adjust** parameters

```
MaterialParameters material
```



**Desired output**

```
float3 reference
```



**“Backward” render**

```
float3 render( /* ... */ )
```

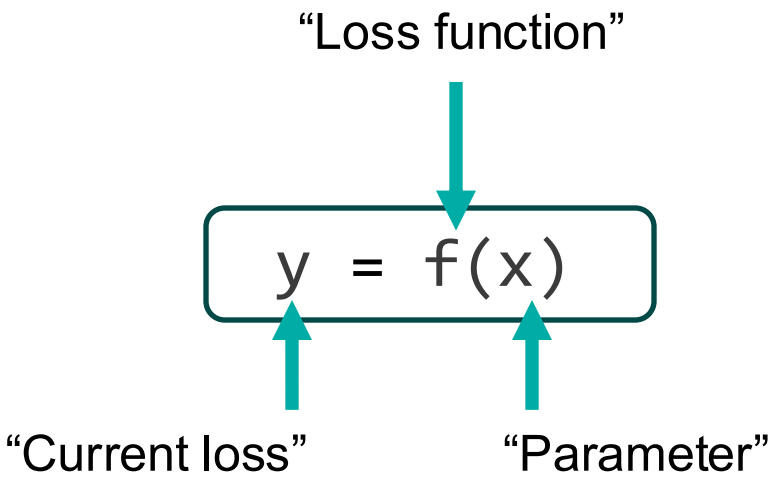
**“Backward” loss**

```
float3 loss( /* ... */ )
```

# THE MATH VIEW: DERIVATIVES



## Inference




## Backward

If I want y to go down, how should I change x?

**Backwards derivative dx:**

```
dx = backwards_deriv_f(x)
```

If you want to change `y` by `delta`…

```
y - delta    ⇒    x - delta * dx
```

…you need to change `x` by `delta * dx`

# DIFFERENTIATION



## Inference

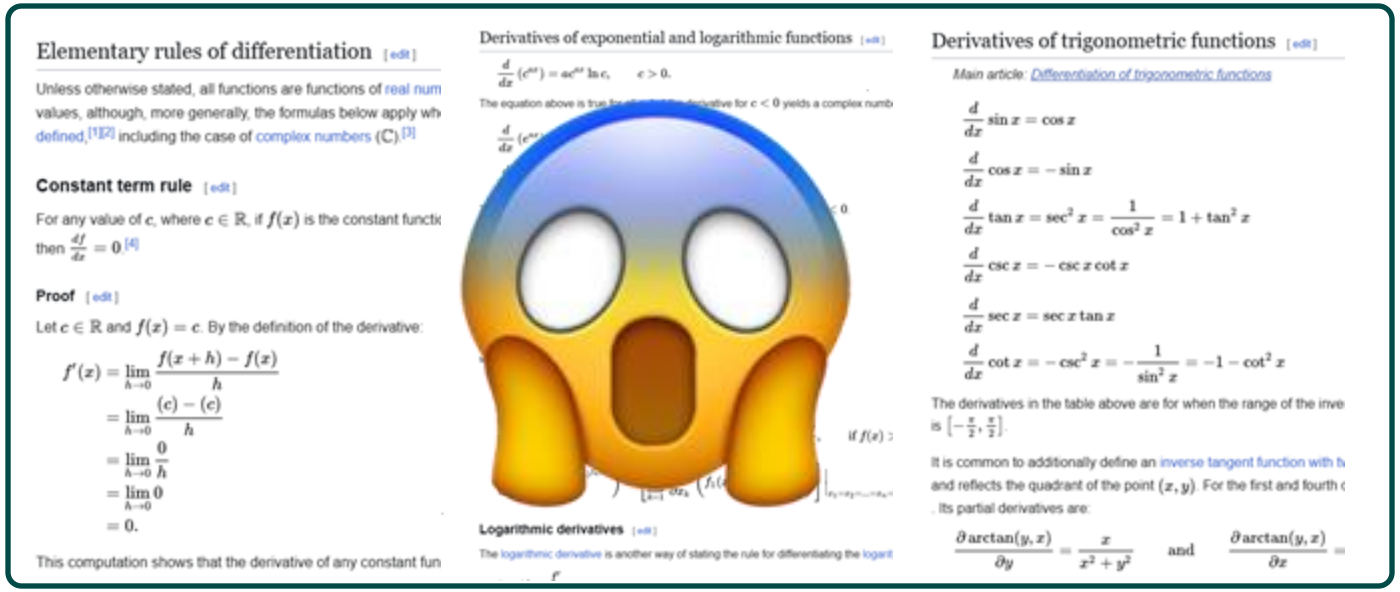
We wrote the inference function:

```
float3 loss(
    int2 pixel,
    float3 reference,
    MaterialParameters material,
    float3 light_dir,
    float3 view_dir)
{
    /* ......  */
}
```

## Backward

…but how do we get the backward function?

In HLSL, we need to **manually derive** it



# AUTOMATIC DIFFERENTIATION



## Inference

We wrote the inference function:

```
float3 loss(
    int2 pixel,
    float3 reference,
    MaterialParameters material,
    float3 light_dir,
    float3 view_dir)
{
    /* ......  */
}
```

## Backward

…but how do we get the backward function?

In Slang, we can let the compiler derive it!

```
bwd_diff(loss)( /* ... */ );
```

# GENERATING THE DERIVATIVES



```
void calculate_grads(
    int2 pixel,
    MaterialParameters material,
    MaterialParameters ref_material)
{
    // Generate random directions
    float3 light_dir = random_direction();
    float3 view_dir = random_direction();

    // Evaluate target value
    float3 reference = eval_reference(ref_material, pixel, light_dir, view_dir);

    // Backpropagate derivatives
    bwd_diff(loss)(pixel, reference, material, light_dir, view_dir, 1);
}
```

# GENERATING THE DERIVATIVES



```
void calculate_grads(
    int2 pixel,
    MaterialParameters material,
    MaterialParameters ref_material)
{
    // Generate random directions
    float3 light_dir = random_direction();
    float3 view_dir = random_direction();

    // Evaluate target value
    float3 reference = eval_reference(ref_material, pixel, light_dir, view_dir);

    // Backpropagate derivatives
    bwd_diff(loss)(pixel, reference, material, light_dir, view_dir, 1);
}
```

# GENERATING THE DERIVATIVES



```
void calculate_grads(
    int2 pixel,
    MaterialParameters material,
    MaterialParameters ref_material)
{
    // Generate random directions
    float3 light_dir = random_direction();
    float3 view_dir = random_direction();

    // Evaluate target value
    float3 reference = eval_reference(ref_material, pixel, light_dir, view_dir);

    // Backpropagate derivatives
    bwd_diff(loss)(pixel, reference, material, light_dir, view_dir, 1);
}
```

# GENERATING THE DERIVATIVES



```
void calculate_grads(
    int2 pixel,
    MaterialParameters material,
    MaterialParameters ref_material)
{
    // Generate random directions
    float3 light_dir = random_direction();
    float3 view_dir = random_direction();

    // Evaluate target value
    float3 reference = eval_reference(ref_material, pixel, light_dir, view_dir);

    // Backpropagate derivatives
    bwd_diff(loss)(pixel, reference, material, light_dir, view_dir, 1);
}
```

# WHERE DO THE DERIVATIVES GO?



## Inference

Read parameters

```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

## Backward

Write gradients

```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

# WHERE DO THE DERIVATIVES GO?



## Inference

Read parameters

```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

## Backward

Write gradients

```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

# WHERE DO THE DERIVATIVES GO?



## Inference

Read parameters

```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

## Backward

Write gradients

```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

# WHERE DO THE DERIVATIVES GO?



## Inference

Read parameters

```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

## Backward

Write gradients

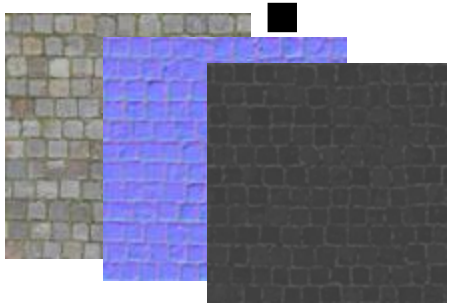
```
struct MaterialParameters
{
    Tensor<float3, 2> albedo;

    RWTensor<float3, 2> albedo_grad;

    float3 get_albedo(int2 pixel)
    {
        return albedo.getv(pixel);
    }

    [BackwardDerivativeOf(get_albedo)]
    void get_albedo_bwd(int2 pixel, float3 grad)
    {
        albedo_grad.setv(pixel, grad);
    }
}
```

# THE “BACKWARD” PHASE



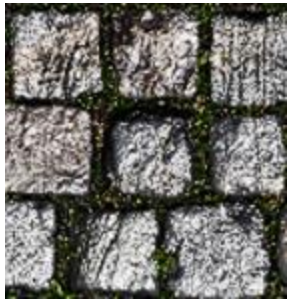
- **Adjust parameters**

```
MaterialParameters material
```



**Desired output**

```
float3 reference
```



**“Backward” render**

```
float3 render( /* ... */ )
```

**“Backward” loss**

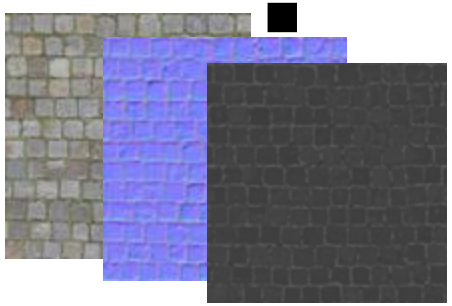
```
float3 loss( /* ... */ )
```

# THE “BACKWARD” PHASE



**Adjust parameters**

```
MaterialParameters material
```



**“Backward” render**

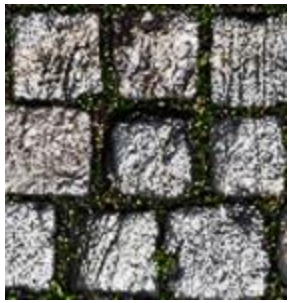
```
float3 render( /* ... */ )
```

**“Backward” loss**

```
float3 loss( /* ... */ )
```

**Desired output**

```
float3 reference
```



# CLOSING THE LOOP: TAKING A STEP



If you want to change `y` by `delta`…

```
y - delta    ⇒    x - delta * dx
```

…you need to change `x` by `delta * dx`

```
void optimizer_step(
    inout float3 mipmap,
    float3 derivative,
    float learning_rate)
{
    mipmap -= learning_rate * derivative;
}
```

# CLOSING THE LOOP: TAKING A STEP



If you want to change y by `delta`…

```
y - delta    ⇒    x - delta * dx
```

…you need to change x by `delta * dx`

```
void optimizer_step(
    inout float3 mipmap,
    float3 mipmap_grad,
    float learning_rate)
{
    mipmap -= learning_rate * mipmap_grad;
}
```

# CLOSING THE LOOP: TAKING A STEP



If you want to change y by `delta`…

```
y - delta    ⇒    x - delta * dx
```

…you need to change x by `delta * dx`

```
void optimizer_step(
    inout float3 mipmap,
    float3 mipmap_grad,
    float learning_rate)
{
    mipmap -= learning_rate * mipmap_grad;
}
```

# CLOSING THE LOOP: TAKING A STEP



If you want to change `y` by `delta`…

`y - delta` ⇒ `x - delta * dx`

…you need to change `x` by `delta * dx`

```
void optimizer_step(
    inout float3 mipmap,
    float3 mipmap_grad,
    float learning_rate)
{
    mipmap -= learning_rate * mipmap_grad;
}
```

# PUTTING IT TOGETHER: GENERATING DERIVATIVES



```
module.calculate_grads(
    material = {
            "albedo": trained_albedo_map,
            "albedo_grad": albedo_grad,
            # ...
    },
    # ...
)
```

```
void calculate_grads(
    MaterialParameters material, /*...*/)
{
    /* ... */

    // Backpropagate derivatives
    bwd_diff(loss)(/* ... */);
}
```

# PUTTING IT TOGETHER: TAKING AN OPTIMIZER STEP



```
module.calculate_grads(
    material = {
            "albedo": trained_albedo_map,
            "albedo_grad": albedo_grad,
            # ...
    },
    # ...
)

module.optimizer_step(
    trained_albedo_map,
    albedo_grad,
    learning_rate)
# ...
```

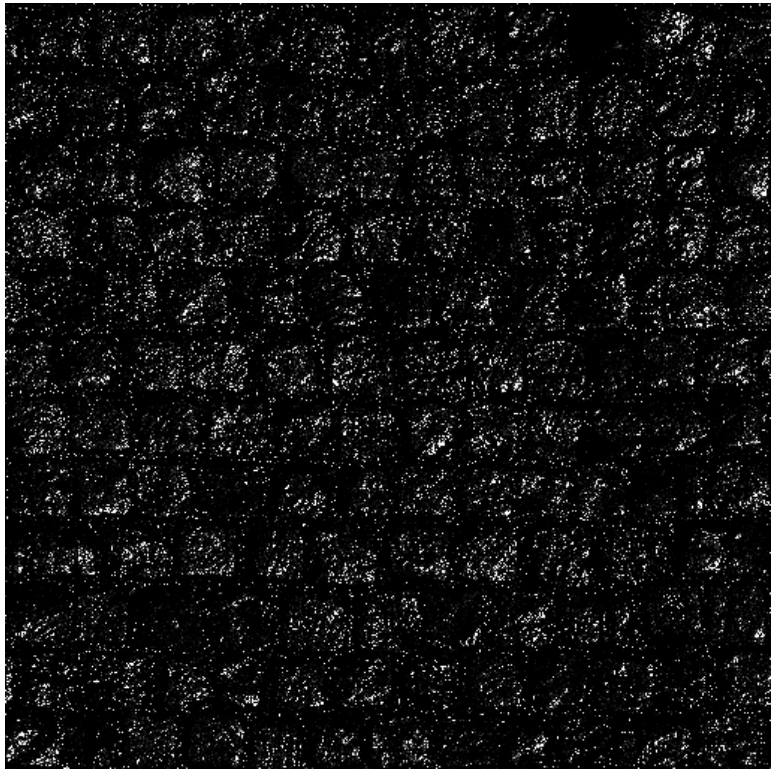
```
void calculate_grads(
    MaterialParameters material, /*...*/)
{
    /* ... */

    // Backpropagate derivatives
    bwd_diff(loss)(/* ... */);
}

void optimizer_step(
    inout float3 mipmap,
    float3 mipmap_grad,
    float learning_rate)
{
    mipmap -= learning_rate * mipmap_grad;
}
```

# LET'S RUN IT!



Reference

Basic Mip-Mapped Textures

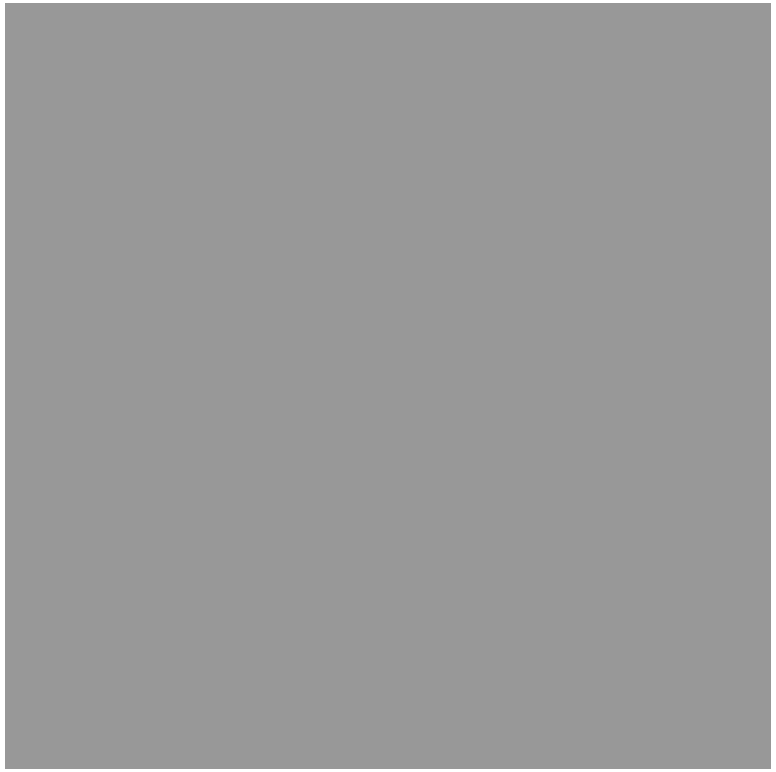


Loss

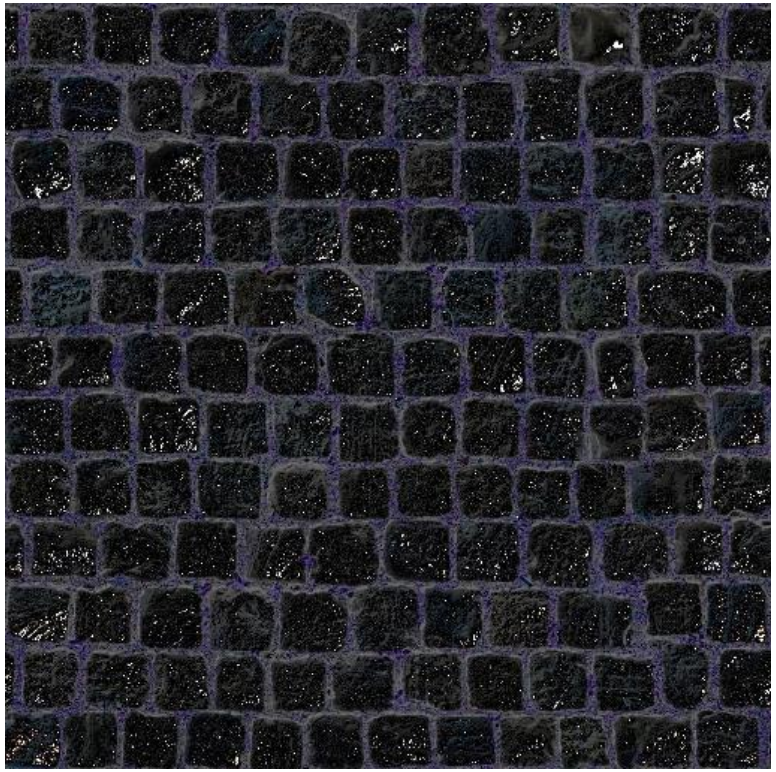
# LET'S RUN IT!



Reference



Optimized Textures



Loss

# SOMETHING IS WRONG WITH OUR DERIVATIVES



```
y - delta   ⇒   x - delta * dx
```

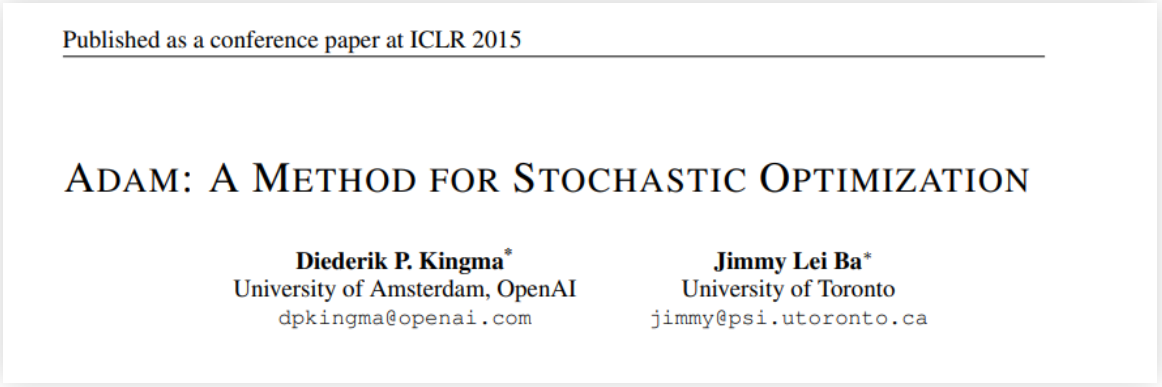
```
dx = backwards_deriv_f(x)
```

```
void calculate_grads(
    int2 pixel,
    MaterialParameters material,
    MaterialParameters ref_material)
{
    // Generate random directions
    float3 light_dir = random_direction();
    float3 view_dir = random_direction();

    // Evaluate target value
    float3 reference = eval_reference(ref_material, pixel, light_dir, view_dir);

    // Backpropagate derivatives
    bwd_diff(loss)(pixel, reference, material, light_dir, view_dir, 1);
}
```

# DEALING WITH NOISY DERIVATIVES



- Our derivatives are noisy. Can we fix it?
- Source of the issue: The optimizer
- Let's swap it out for one that can handle noise!
  - Usual approach: Exponential moving average of gradients
- Even better: Adam optimizer



Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

**Diederik P. Kingma***
University of Amsterdam, OpenAI
dpkingma@openai.com

**Jimmy Lei Ba***
University of Toronto
jimmy@psi.utoronto.ca

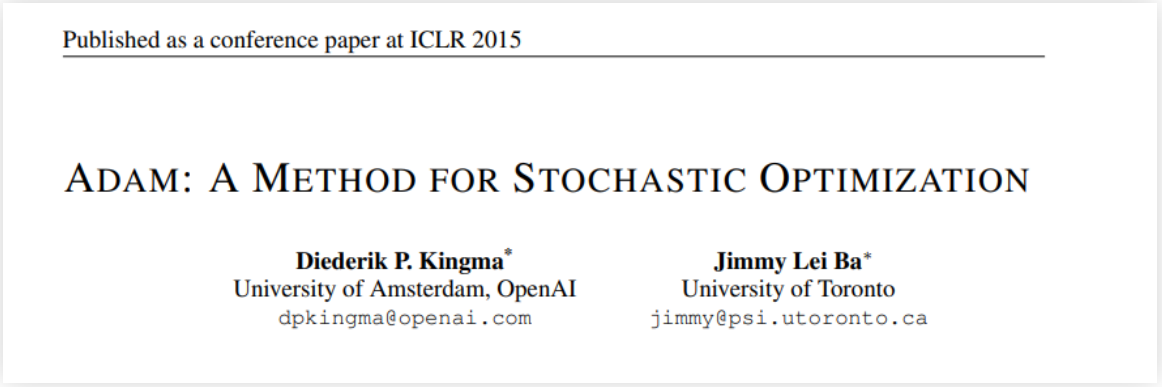
## Gradient Descent

```
void optimizer_step(
    inout float3 mipmap,
    float3 mipmap_grad,
    float learning_rate)
{
    mipmap -= learning_rate * mipmap_grad;
}
```

# DEALING WITH NOISY DERIVATIVES



- Our derivatives are noisy. Can we fix it?
- Source of the issue: The optimizer
- Let's swap it out for one that can handle noise!
  - Usual approach: Exponential moving average of gradients
- Even better: Adam optimizer



Published as a conference paper at ICLR 2015

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

**Diederik P. Kingma***
University of Amsterdam, OpenAI
dpkingma@openai.com

**Jimmy Lei Ba***
University of Toronto
jimmy@psi.utoronto.ca

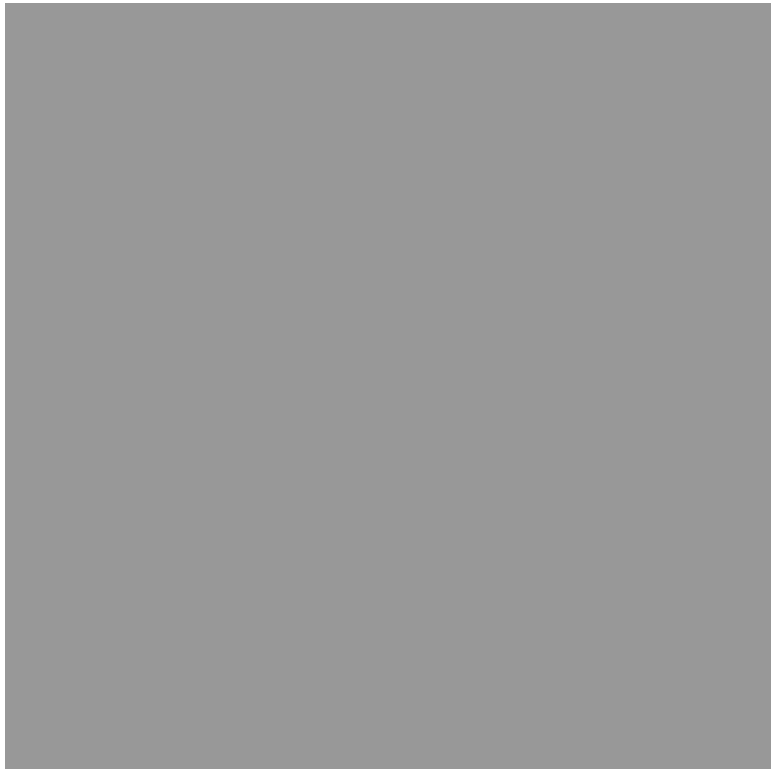
## Adam Optimizer

```
void optimizer_step(
    inout float3 mipmap,
    inout float3 mipmap_grad,
    inout float3 mean,
    inout float3 variance,
    float learning_rate,
    int iteration)
{
    mean = lerp(mipmap_grad, mean, 0.9f);
    variance = lerp(mipmap_grad * mipmap_grad, variance, 0.999f);

    float3 mHat = mean / (1.0f - pow(beta1, iteration));
    float3 vHat = variance / (1.0f - pow(beta2, iteration));
    float3 step = mHat / (sqrt(vHat) + 1e-8f);

    mipmap -= learning_rate * step;
}
```

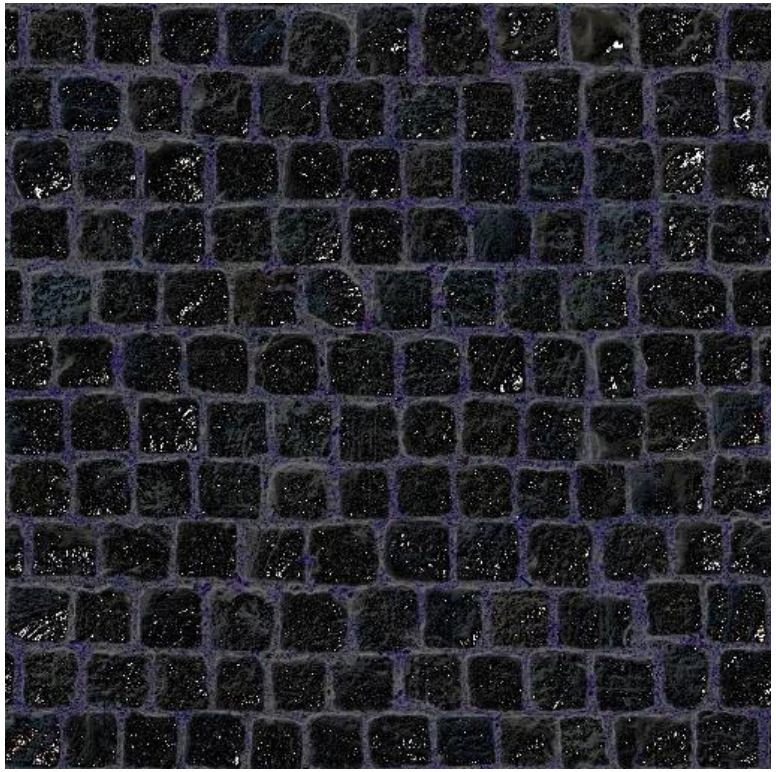
# LET’S RUN IT!



Reference



Adam Optimized Textures



Loss

# SUMMARY

# SUMMARY: THE CLASSICAL APPROACH



- We started with a problem that is somewhat tricky
  - Generate texture mip maps without changing the look of the material
- There are techniques developed for this (e.g. Toksvig), but…
  - They are developed for a specific class of material
  - Your material doesn’t fit in this mold? You’ll have to invent something new
  - That’s a lot of work!

# SUMMARY: NEURAL SHADING



- Alternative: Write down the error, then make it smaller

- In Slang: The bulk of the code you write *is* the inference code
  - I.e., the code you want to ship!
- You can change the inference code and don't need to come up with e.g. a new mip-mapping method

This is the benefit of neural shading:

- Problems you don't know how to solve well (yet), may now be solvable
- With the right tools, you don't have to spend a lot of effort to do this

SIGGRAPH 2025
Vancouver+ 10-14 August

# YOUR FIRST MLP IN A SHADER

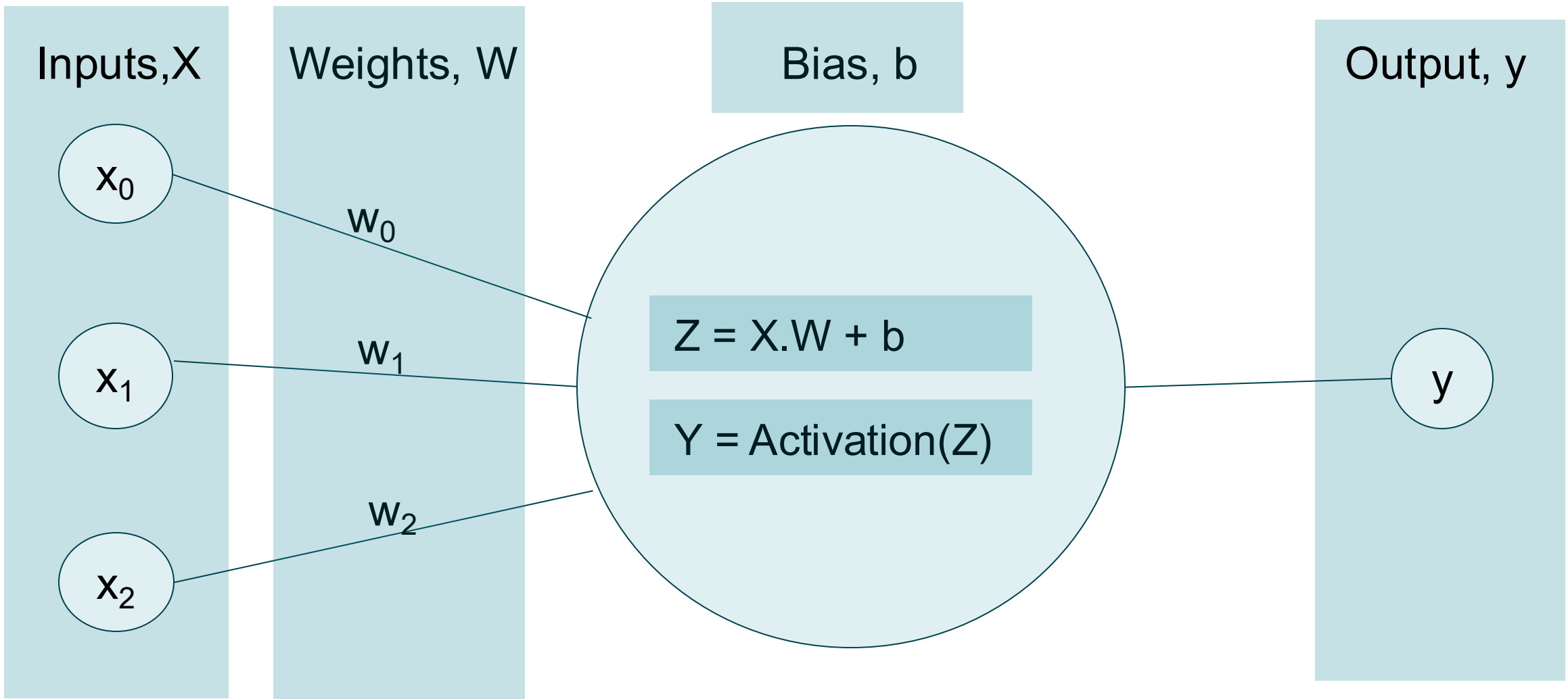
# DOES IT HAVE TO BE PIXELS?



- Training mipmap levels shows we can *learn* the pixels of a texture, based on a desired output.
- Can we *learn* a neural network that generates the pixels of a texture?

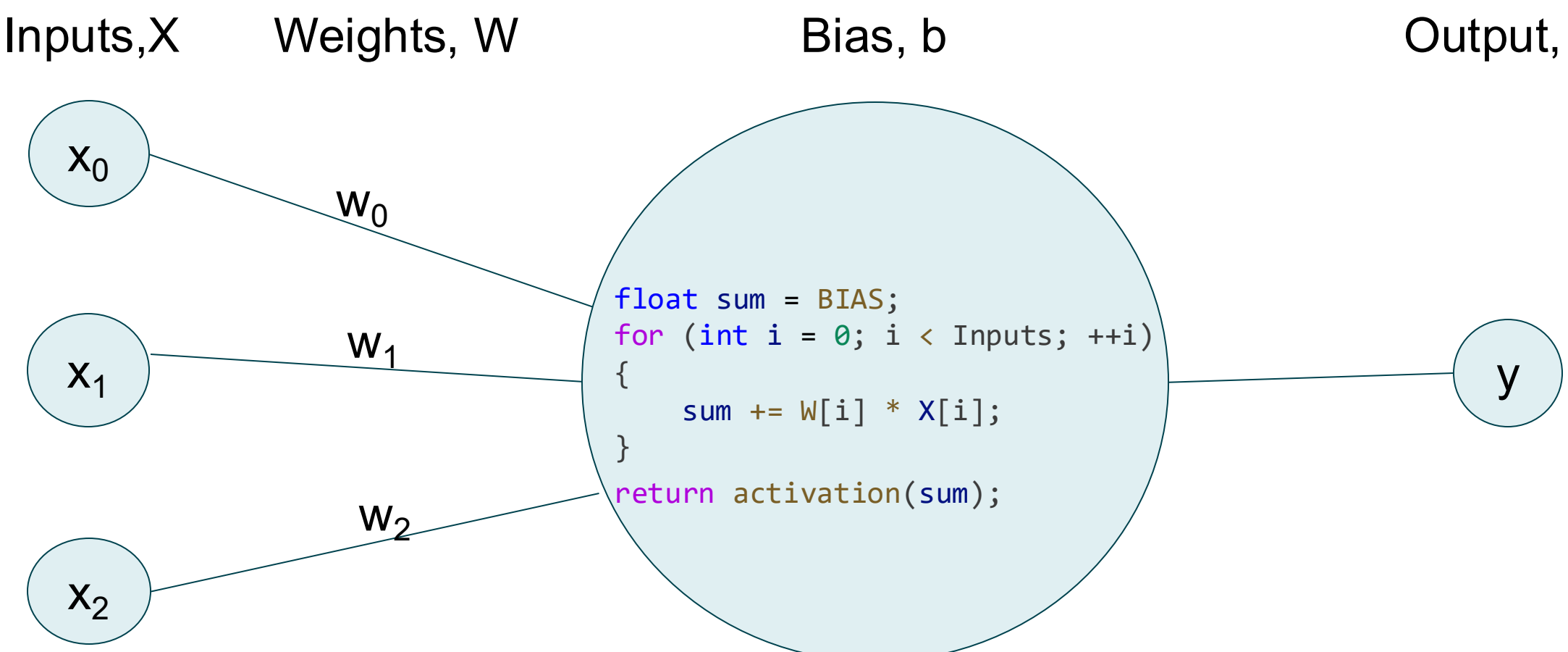
# THE BUILDING BLOCK





# THE BUILDING BLOCK

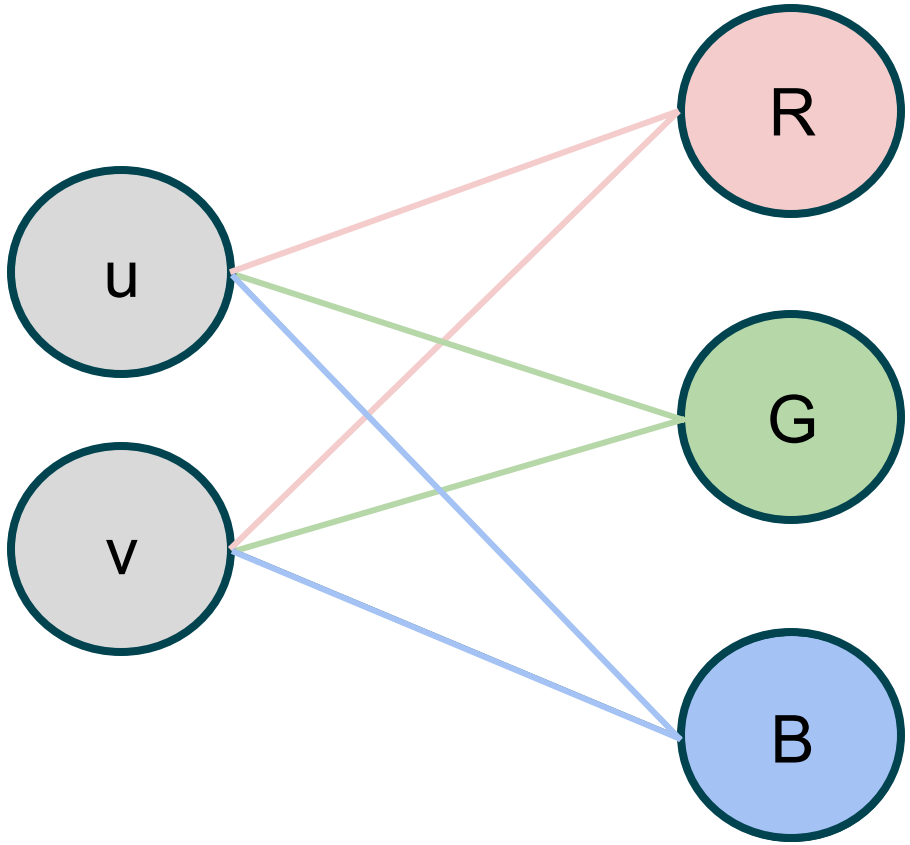




# THE SIMPLEST TEXTURE NETWORK



- Inputs: Texture coordinate (uv)
- Outputs: Color channels (rgb)
- 6 weights (2 inputs * 3 outputs)
- 3 biases
- **9 parameters total**



# NETWORK PARAMETERS



```
struct NetworkParameters<int Inputs, int Outputs>
{
    RWTensor<float, 1> biases;
    RWTensor<float, 2> weights;

    AtomicTensor<float, 1> biases_grad;
    AtomicTensor<float, 2> weights_grad;

    [Differentiable]
    float get_bias(int neuron)
    { /* ... */ }

    [Differentiable]
    float get_weight(int neuron, int input)
    { /* ... */ }

    [BackwardDerivativeOf(get_bias)]
    void get_bias_bwd(int neuron, float grad)
    { /* ... */ }

    [BackwardDerivativeOf(get_weight)]
    void get_weight_bwd(int neuron, int input, float grad)
    { /* ... */ }
}
```

```
class NetworkParameters(spy.InstanceList):
    def __init__(self, inputs: int, outputs: int):
        super().__init__( ... )
        self.inputs = inputs
        self.outputs = outputs

        # Biases and weights for the layer.
        self.biases = spy.Tensor.from_numpy( ... )
        self.weights = spy.Tensor.from_numpy( ... )

        # Gradients for the biases and weights.
        self.biases_grad = spy.Tensor.zeros_like(self.biases)
        self.weights_grad = spy.Tensor.zeros_like(self.weights)

# Calls the Slang 'optimize' function for biases and weights
def optimize(self, ... ):
    module.optimize1( ... )
    module.optimize1( ... )
```

# NETWORK PARAMETERS



## Evaluating 1 layer

```
struct NetworkParameters<int Inputs, int Outputs>
{
    // ... Fields and accessors here ...

    // Evaluate all neurons in the layer
    [Differentiable]
    float[Outputs] forward(float[Inputs] x)
    {
        float[Outputs] y;
        [ForceUnroll]
        for (int row = 0; row < Outputs; ++row)
        {
            // Sum up bias plus weighted inputs for 1 neuron
            var sum = get_bias(row);
            [ForceUnroll]
            for (int col = 0; col < Inputs; ++col)
                sum += get_weight(row, col) * x[col];
            y[row] = sum;
        }

        return y;
    }

}
```

## Evaluating the network

```
struct Network {
    NetworkParameters<2, 3>  layer;

    // Evaluate the layer and output activations
    [Differentiable]
    float3 eval(no_diff float2 uv)
    {
        // Convert uv to array of 2 inputs
        float inputs[2] = {uv.x, uv.y};

        // Feed inputs to layer, get outputs back
        float output0[3] = layer.forward(inputs);

        // Apply activation function to each output
        [ForceUnroll]
        for (int i = 0; i < 3; ++i)
            output0[i] = activation(output0[i]);

        // Return the output (to be used an RGB colour)
        return float3(output0[0], output0[1], output0[2]);
    }
}
```

# RENDER SOMETHING



Reference

Output
(untrained!)

```
[Differentiable]
float3 render(int2 pixel, int2 resolution, Network network)
{
    float2 uv = (float2(pixel) + 0.5f) / float2(resolution);
    return network.eval(uv);
}
```

Loss
(a lot!)

```
[Differentiable]
float3 loss(int2 pixel, int2 resolution, no_diff float3 reference, Network network)
{
    float3 color = render(pixel, resolution, network);
    float3 error = color - reference;
    return error * error; // Squared error
}
```

# TRAINING



Back propagate gradients of loss function to the network's weights and biases

```
void calculate_grads(int2 pixel,
                     int2 resolution,
                     float3 reference,
                     Network network)
{
    bwd_diff(loss)(pixel, resolution,
                   reference, network, 1.0f);
}
```

Calculate gradients, then call optimize to apply them

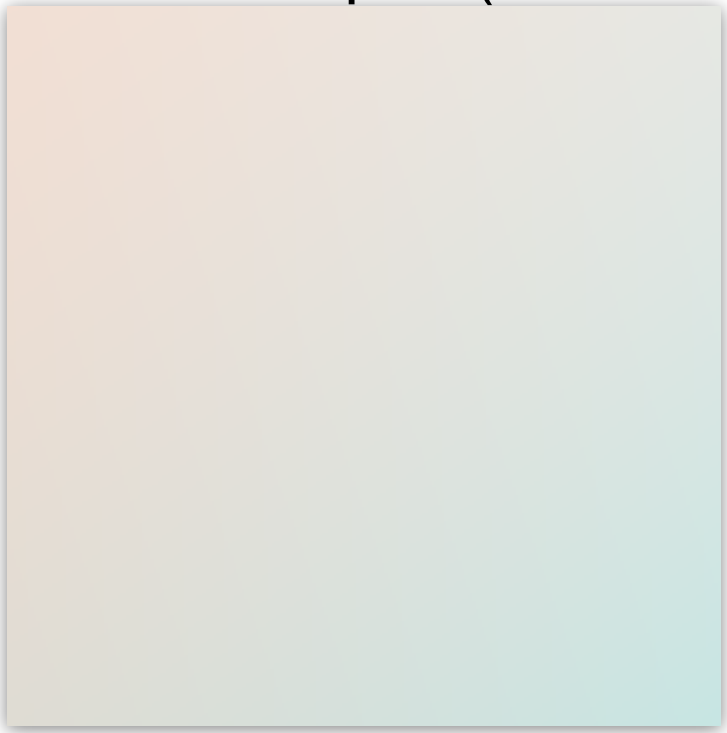
```
learning_rate = 0.001

for i in range(50):
    module.calculate_grads(
        pixel = spy.call_id(),
        resolution = res,
        reference = image,
        network = network)
    optimize_counter += 1

    network.optimize(learning_rate, optimize_counter)
```

# RESULT!



Reference

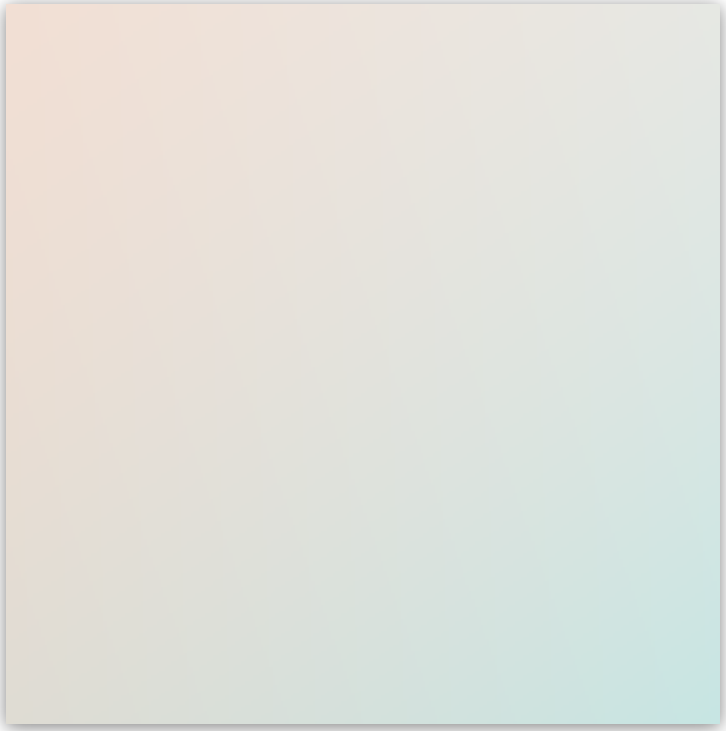
Output :(



# RESULT!



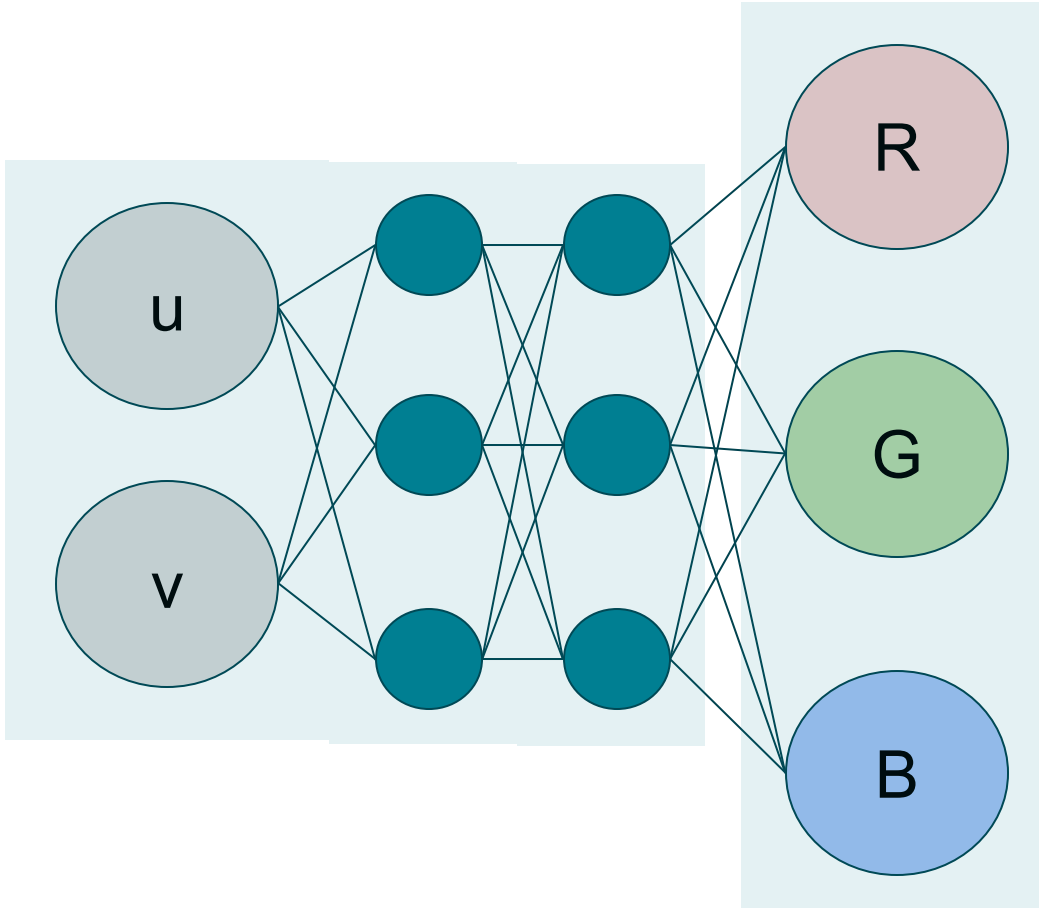
- **Just 9 parameters** and a bit of maths!
- Simple gradient fill is probably the best solution



# HIDDEN LAYERS



- Inputs: Texture coordinate (uv)
- Outputs: Color channels (rgb)
- For 2 * 32 node hidden layers
  - 67 biases
  - 1184 weights
- **1251 parameters total**



# IMPLEMENTATION



## Slang: Connect the layer inputs/outputs

```
struct Network {
    NetworkParameters<2, 32>  layer0;
    NetworkParameters<32, 32> layer1;
    NetworkParameters<32, 3>  layer2;

    [Differentiable]
    float3 eval(no_diff float2 uv)
    {
        float inputs[2] = {uv.x, uv.y};
        float output0[32] = layer0.forward(inputs);
        [ForceUnroll]
        for (int i = 0; i < 32; ++i)
            output0[i] = activation(output0[i]);

        float output1[32] = layer1.forward(output0);
        [ForceUnroll]
        for (int i = 0; i < 32; ++i)
            output1[i] = activation(output1[i]);

        float output2[3] = layer2.forward(output1);
        [ForceUnroll]
        for (int i = 0; i < 3; ++i)
            output2[i] = activation(output2[i]);

        return float3(output2[0], output2[1], output2[2]);
    }
}
```

## Python: allocate/optimize more layers

```
class Network(spy.InstanceList):
    def __init__(self):
        super().__init__(module["Network"])
        self.layer0 = NetworkParameters(2,32)
        self.layer1 = NetworkParameters(32,32)
        self.layer2 = NetworkParameters(32,3)

    # Calls the Slang 'optimize' function for the layer.
    def optimize(self, learning_rate: float, optimize_counter: int):
        self.layer0.optimize(learning_rate, optimize_counter)
        self.layer1.optimize(learning_rate, optimize_counter)
        self.layer2.optimize(learning_rate, optimize_counter)
```

# RESULT!



Reference

Better!

SIGGRAPH 2025
Vancouver+ 10-14 August

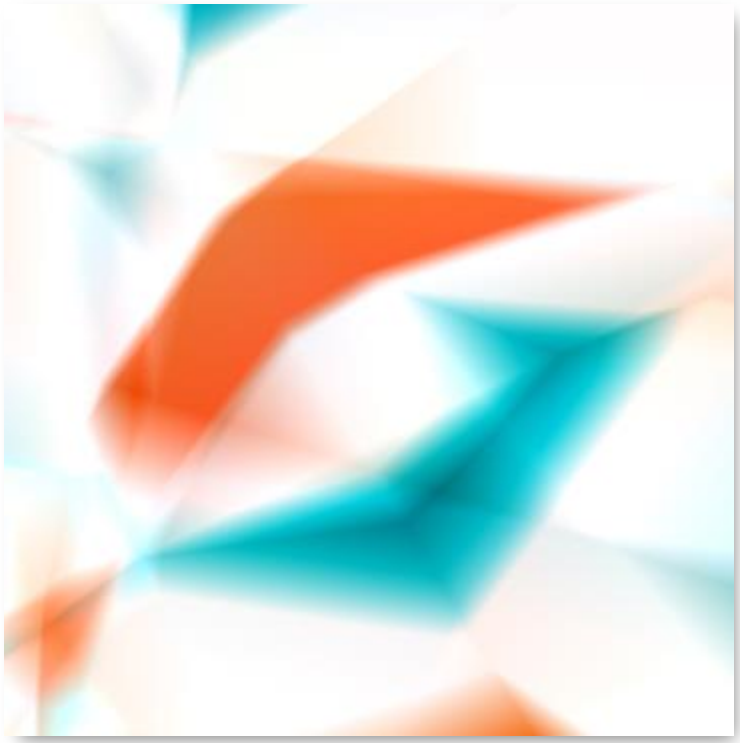
# LEARNING A NEURAL SHADER

# RESULT!



Reference

Better!

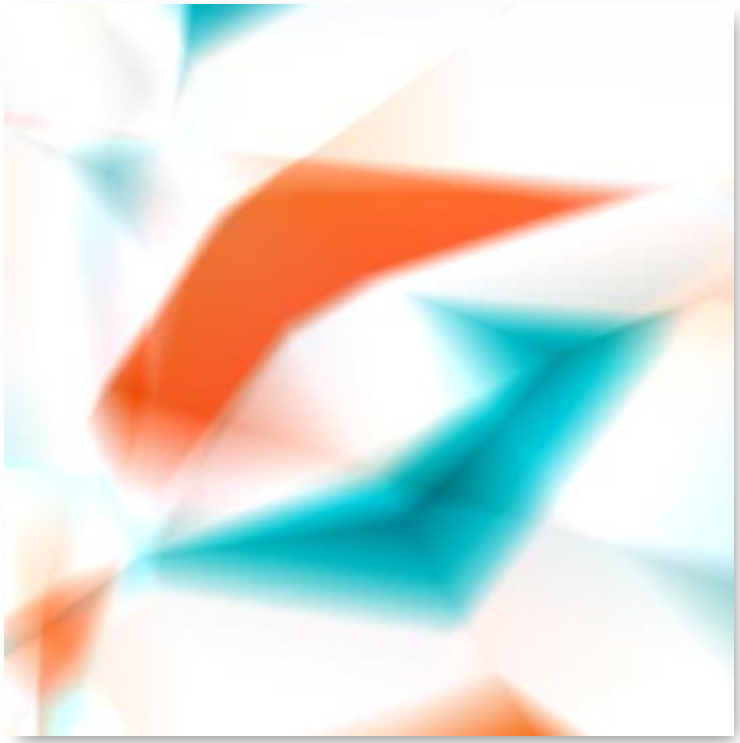


This does not look very good…

# ADDING MORE PARAMETERS?



Reference

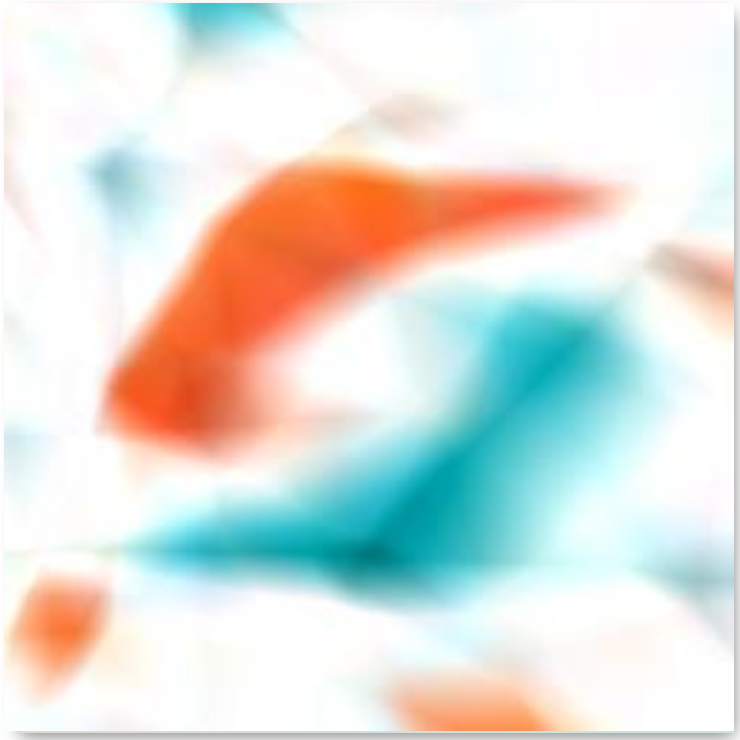


3 layers, 32 neurons

# ADDING MORE PARAMETERS?



Reference



3 layers, 64 neurons

# ADDING MORE PARAMETERS?



Reference

4 layers, 64 neurons

# ADDING MORE PARAMETERS?



- Given enough parameters, this MLP will *eventually* work
  - MLPs can approximate *anything*!
- …but we only have a finite amount of time per frame!
- Is this even practical for real-time?

4 layers, 64 neurons

# ADDING MORE PARAMETERS?

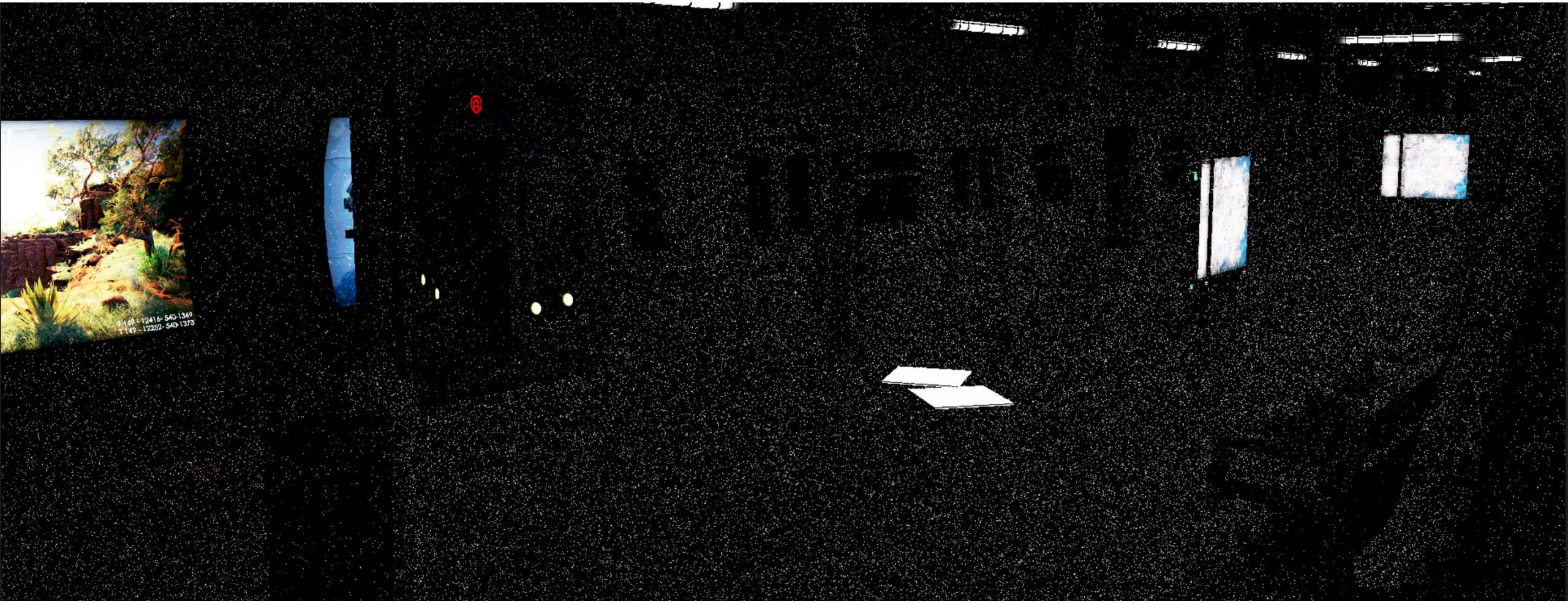


- Given enough parameters, this MLP will *eventually* work
  - MLPs can approximate *anything*!
- …but we only have a finite amount of time per frame!
- Is this even practical for real-time?
- This sounds familiar!

4 layers, 64 neurons

# “MONTE CARLO RAYTRACING CAN RENDER ANYTHING”





“…eventually”

# …WE JUST NEED TO BE SMART ABOUT USING IT



# …WE JUST NEED TO BE SMART ABOUT USING IT

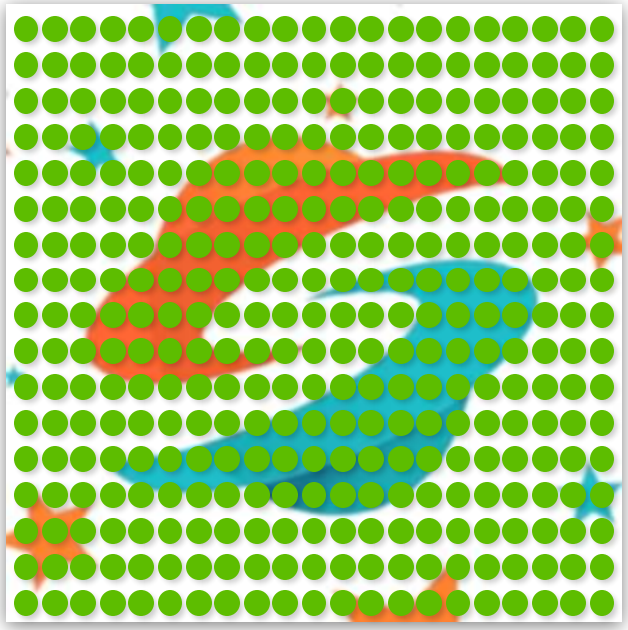


- Any “universal” tool needs careful use if we care about efficiency
- For ray tracing, we have DLSS, ReSTIR, path guiding, …
- Let’s talk about the machine learning tricks we can use for neural shading!

# SPEEDING UP TRAINING



- Training is quite slow
- We are evaluating the loss for every pixel
- We don’t need to!



# SPEEDING UP TRAINING



- Training is quite slow
- We are evaluating the loss for every pixel
- We don’t need to!

# SPEEDING UP TRAINING

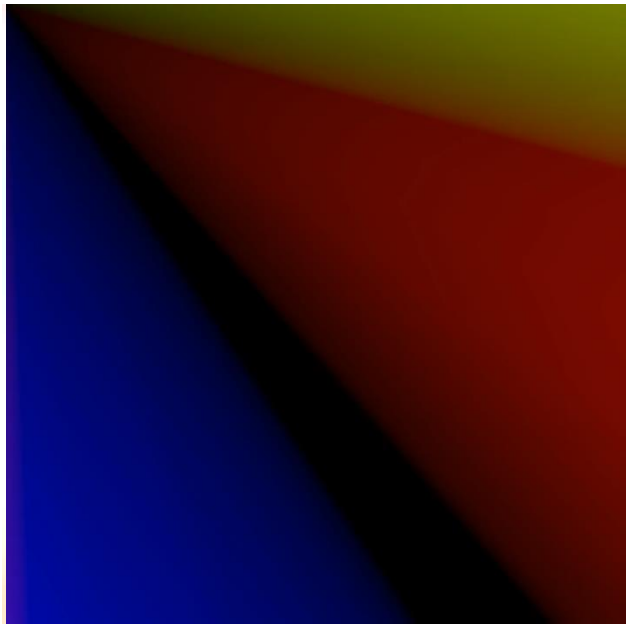


- Training is quite slow
- We are evaluating the loss for every pixel
- We don’t need to!
- Good baseline: Randomly jittered points

# SPEEDING UP TRAINING



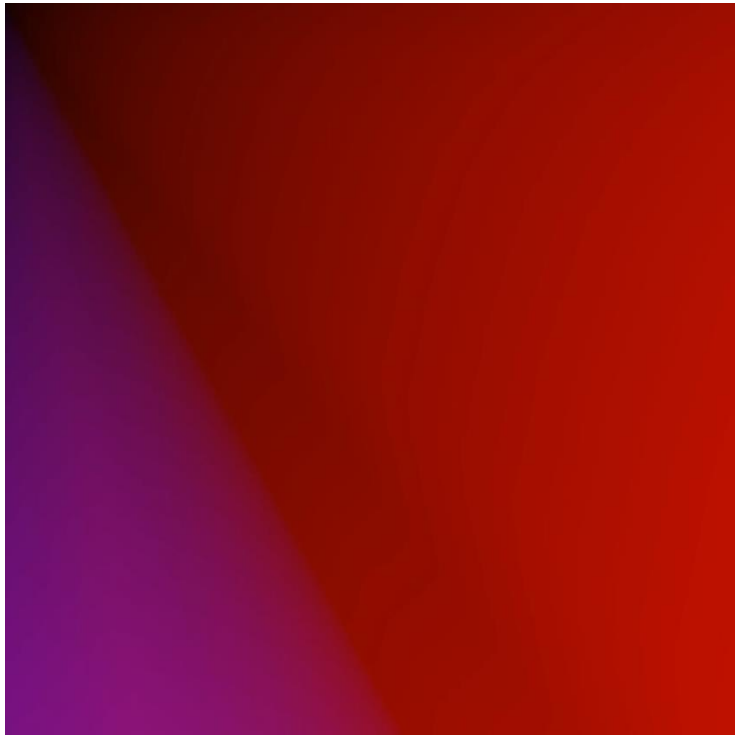
- Training is quite slow
- We are evaluating the loss for every pixel
- We don’t need to!
- Good baseline: Randomly jittered points
- Extra credit: Slowly increase batch size



# A CLOSER LOOK AT ACTIVATIONS



- Sometimes training gets “stuck”



3 layers, 32 neurons

# A CLOSER LOOK AT ACTIVATIONS



- Sometimes training gets “stuck”
- This is because of our choice of activation function:

```
for (int i = 0; i < 32; ++i)
    output0[i] = activation(output0[i]);
/* ... */
for (int i = 0; i < 32; ++i)
    output1[i] = activation(output1[i]);
/* ... */
for (int i = 0; i < 3; ++i)
    output2[i] = activation(output2[i]);
```

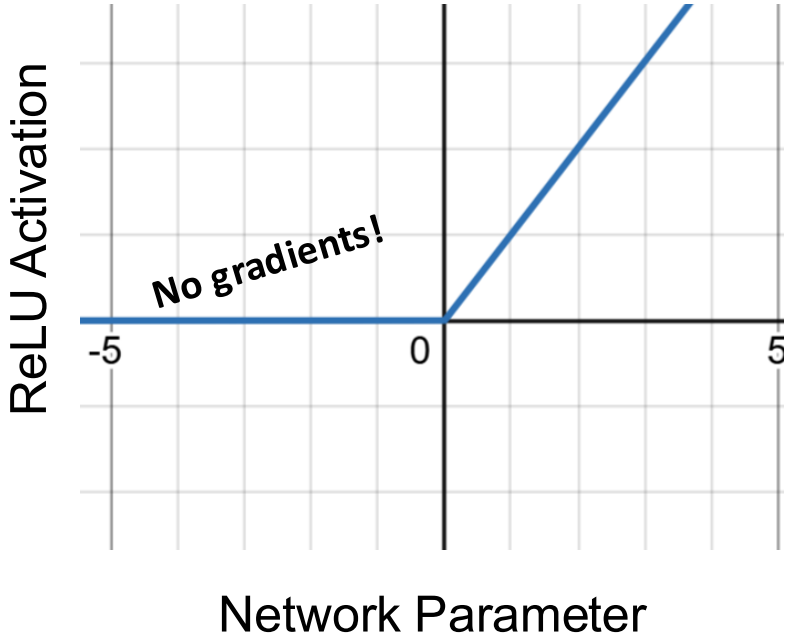
```
float activation(float x)
{
    return max(x, 0.0f);
}
```

3 layers, 32 neurons

# A CLOSER LOOK AT ACTIVATIONS



- This is a “ReLU” activation
- When the output is negative, the neuron receives no gradients
- This can cause small networks to get “stuck”

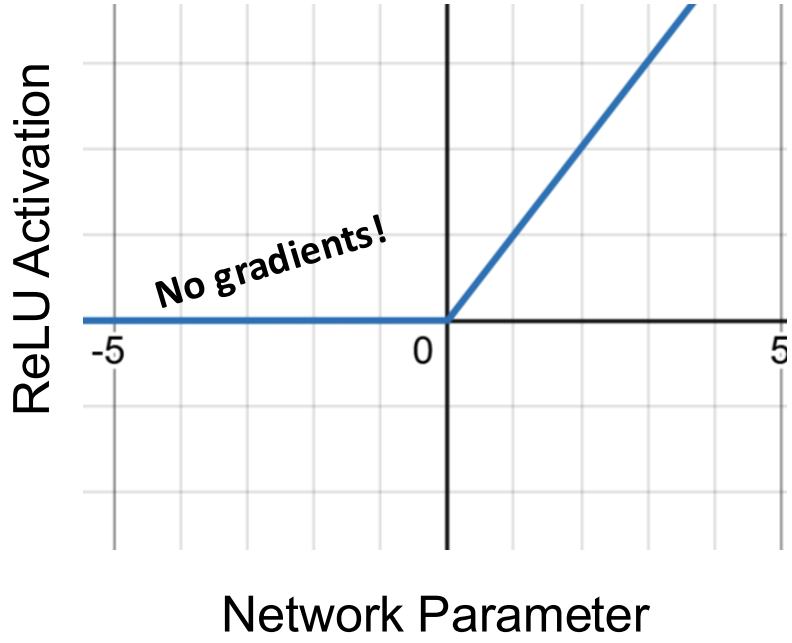


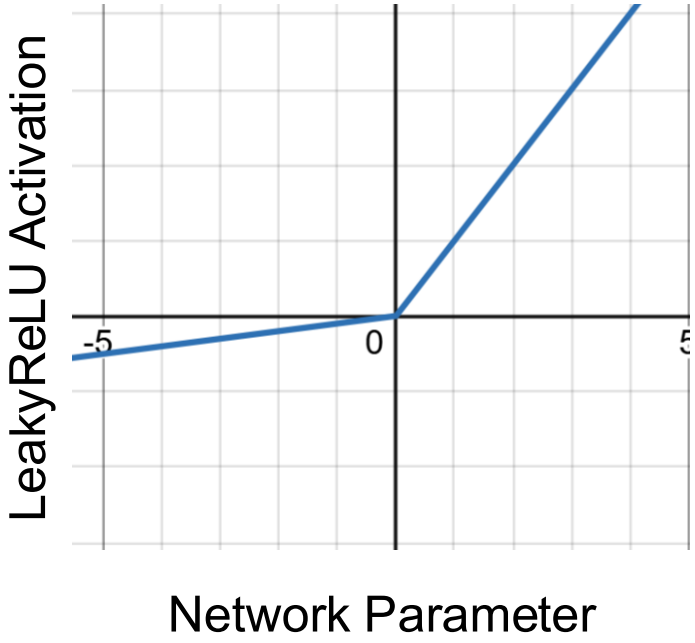
# A CLOSER LOOK AT ACTIVATIONS



- This is a “ReLU” activation
- When the output is negative, the neuron receives no gradients
- A “leaky” activation can help:

```
float leakyReLU(float x)
{
    if (x > 0.0f)
        return x;
    else
        return x * 0.01f;
}
```

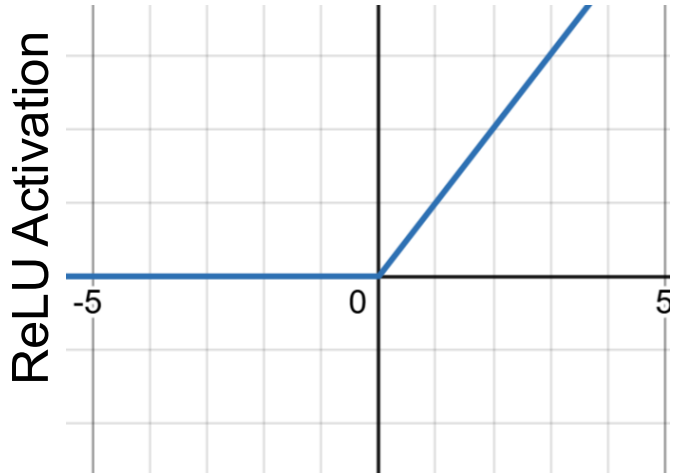


# A CLOSER LOOK AT ACTIVATIONS



- Why does our output look like it's made of triangles?
- A ReLU can only produce piecewise linear outputs
- We can do better by using a smoother activation

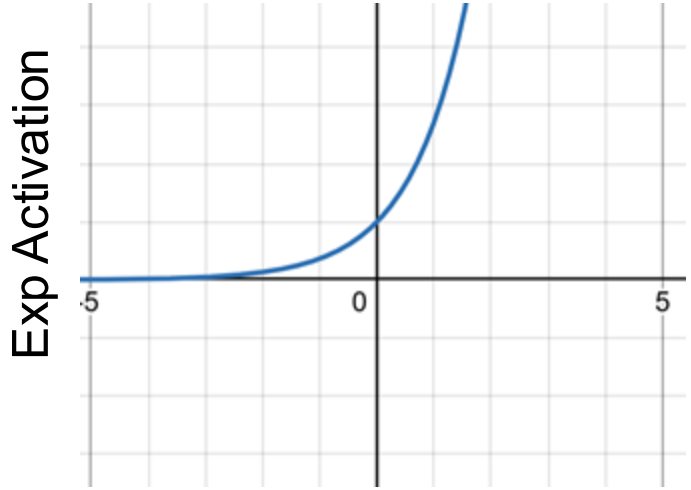



3 layers, 32 neurons

# A CLOSER LOOK AT ACTIVATIONS



- Why does our output look like it's made of triangles?
- A ReLU can only produce piecewise linear outputs
- We can do better by using a smoother activation
- Let's try an exponential:

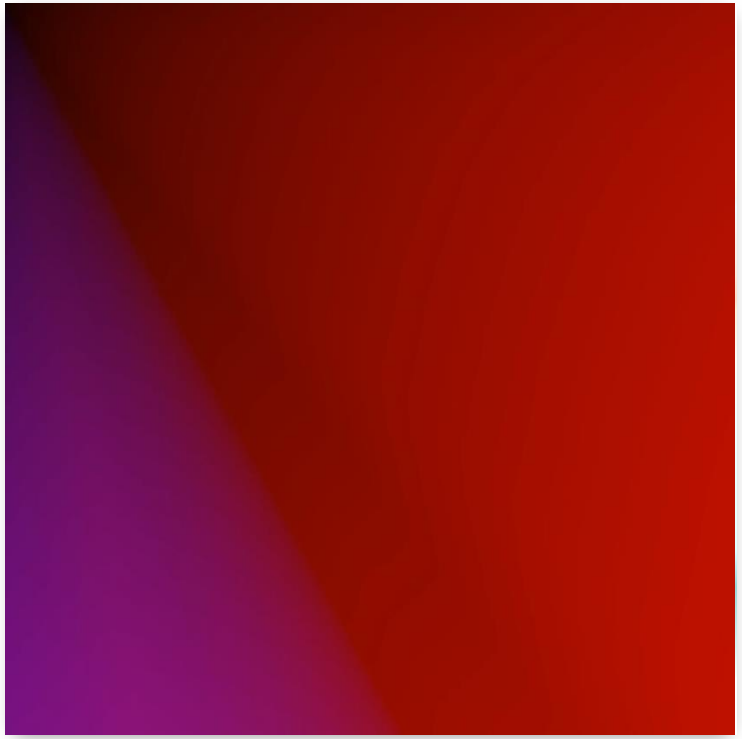



3 layers, 32 neurons

# A CLOSER LOOK AT ACTIVATIONS



```
for (int i = 0; i < 32; ++i)
    output0[i] = activation(output0[i]);
/* ... */
for (int i = 0; i < 32; ++i)
    output1[i] = activation(output1[i]);
/* ... */
for (int i = 0; i < 3; ++i)
    output2[i] = activation(output2[i]);
```



3 layers, 32 neurons

# A CLOSER LOOK AT ACTIVATIONS



```
for (int i = 0; i < 32; ++i)
    output0[i] = leakyReLU(output0[i]);
/* ... */
for (int i = 0; i < 32; ++i)
    output1[i] = leakyReLU(output1[i]);
/* ... */
for (int i = 0; i < 3; ++i)
    output2[i] = exp(output2[i]);
```

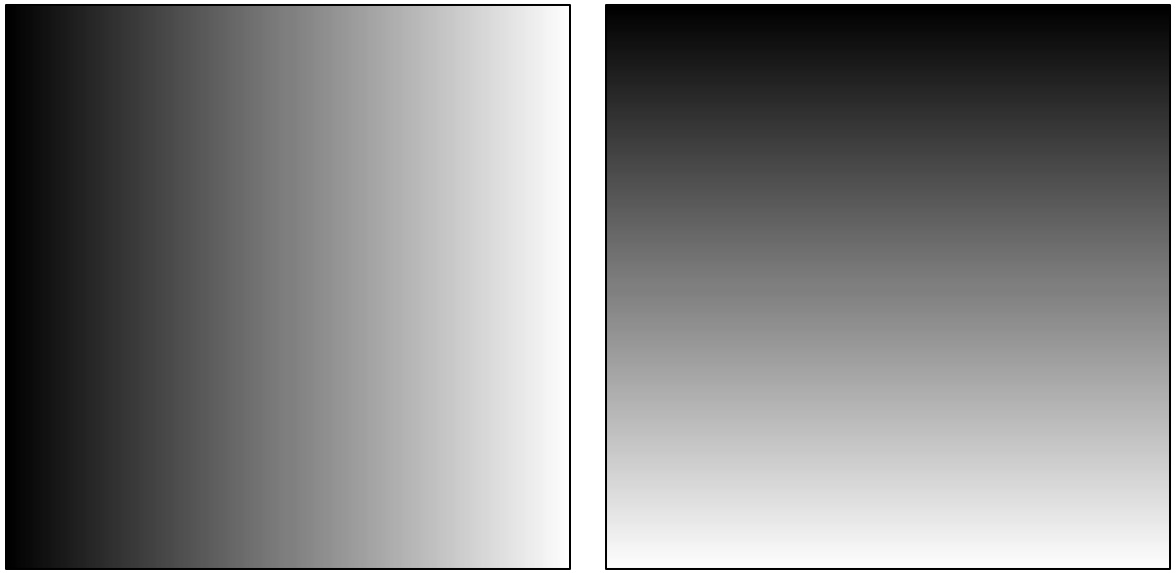


3 layers, 32 neurons

# GIVING BETTER INPUTS



- This is the task we're asking the model to solve:



”Image UVs”

“Image”

- i.e. “transform a gradient into an image”

# GIVING BETTER INPUTS



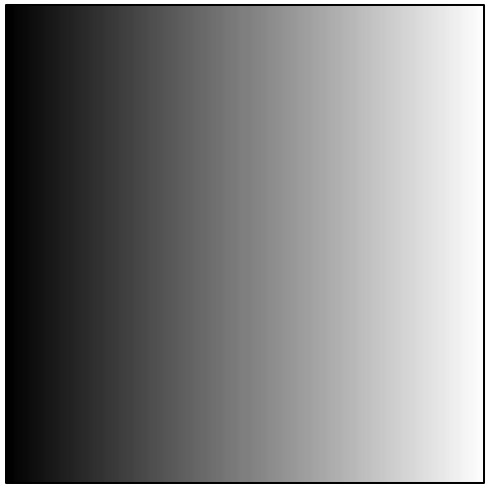
- This isn't usually how we do image representation!
- Image codecs use much more complex building blocks than gradients
  - E.g. cosines (JPEG), wavelets, BC6/BC7 blocks, …
- Given enough parameters, the network will eventually learn something similar
- For a small network, that's wasteful
  - Why learn what you already know?
- Let's try passing some JPEG inspired inputs!
  - "Extract" our domain knowledge from the network into a fixed-function input

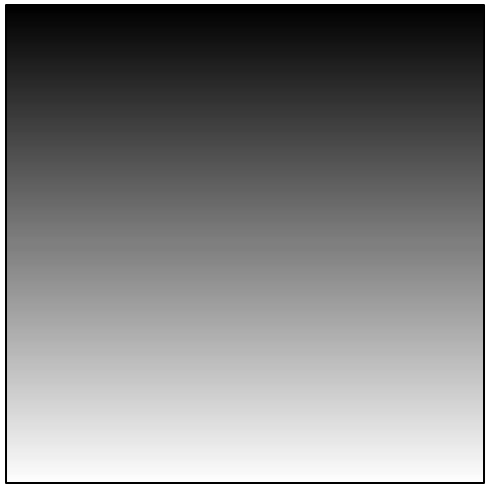
# GIVING BETTER INPUTS



“Frequency encoding”:

- Instead of passing u and v to the network…





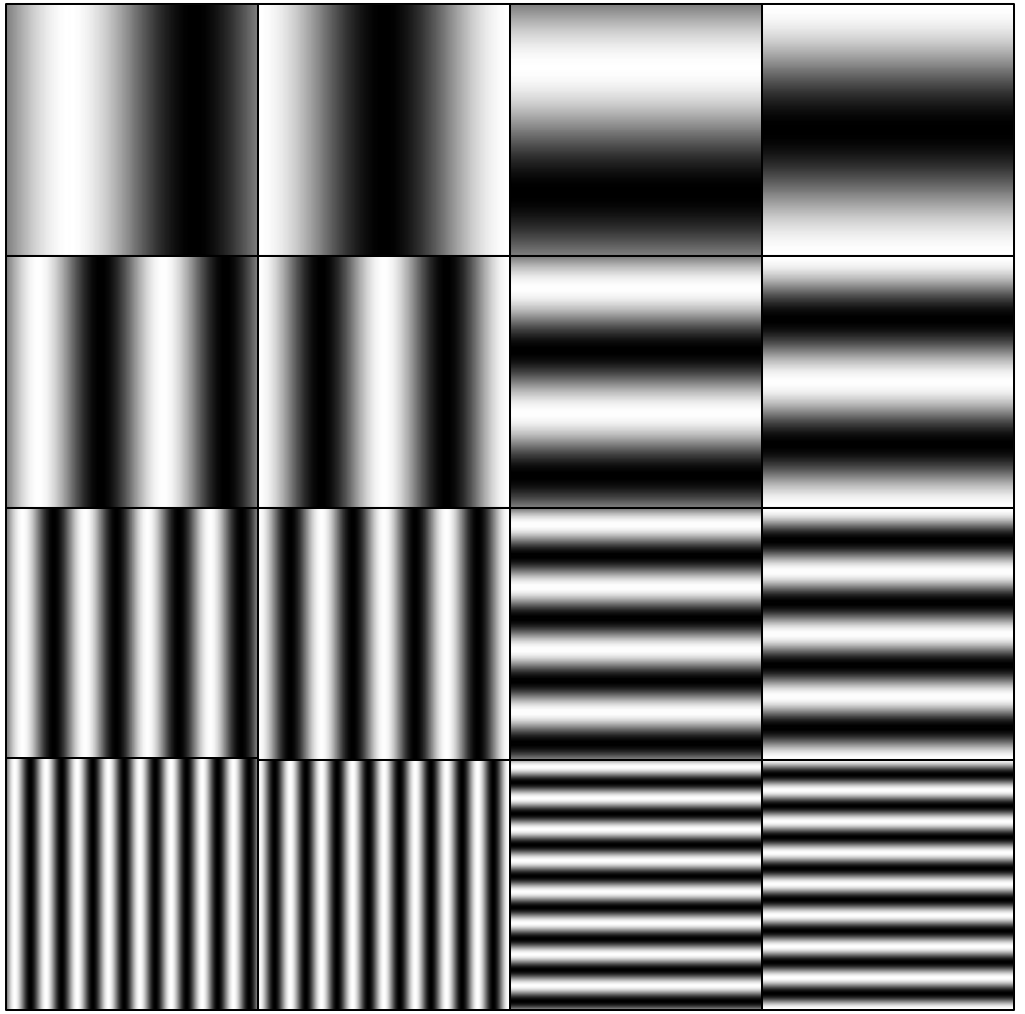
# GIVING BETTER INPUTS



“Frequency encoding”:

- Instead of passing u and v to the network…
- Pass

```
sin(u),  cos(u),  sin(v),  cos(v),
sin(2u), cos(2u), sin(2v), cos(2v),
sin(4u), cos(4u), sin(4v), cos(4v),
…
```



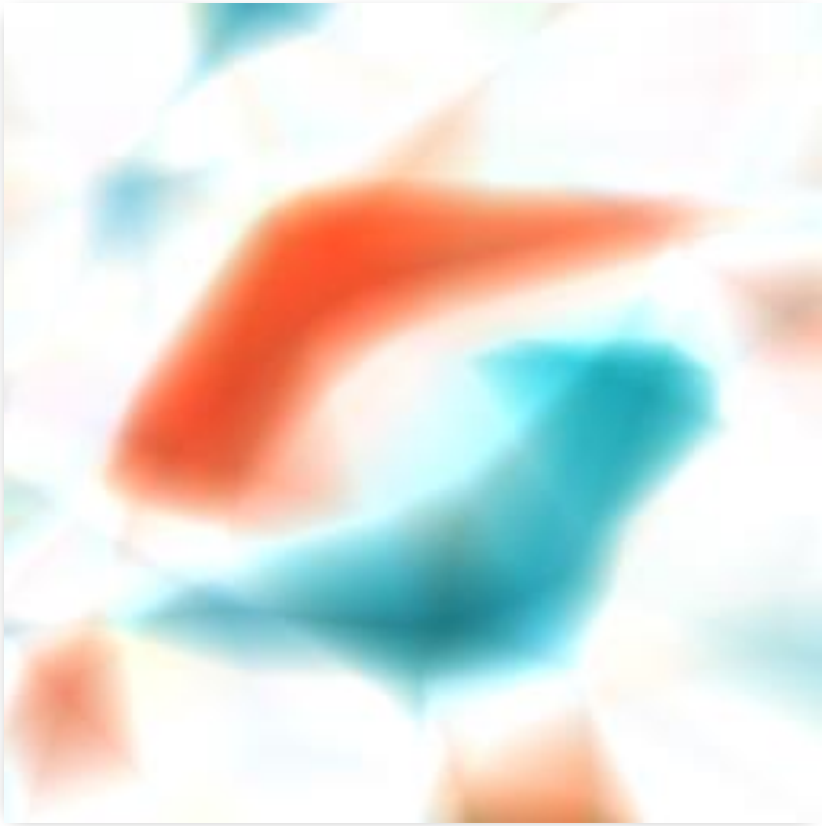
# GIVING BETTER INPUTS



“Frequency encoding”:

- Instead of passing u and v to the network…
- Pass

```
sin(u),  cos(u),  sin(v),  cos(v),
sin(2u), cos(2u), sin(2v), cos(2v),
sin(4u), cos(4u), sin(4v), cos(4v),
…
```



No Input Encoding

# GIVING BETTER INPUTS



“Frequency encoding”:

- Instead of passing u and v to the network…
- Pass

```
sin(u),  cos(u),  sin(v),  cos(v),
sin(2u), cos(2u), sin(2v), cos(2v),
sin(4u), cos(4u), sin(4v), cos(4v),
…
```

Frequency Encoding

# ADDING MORE PARAMETERS?



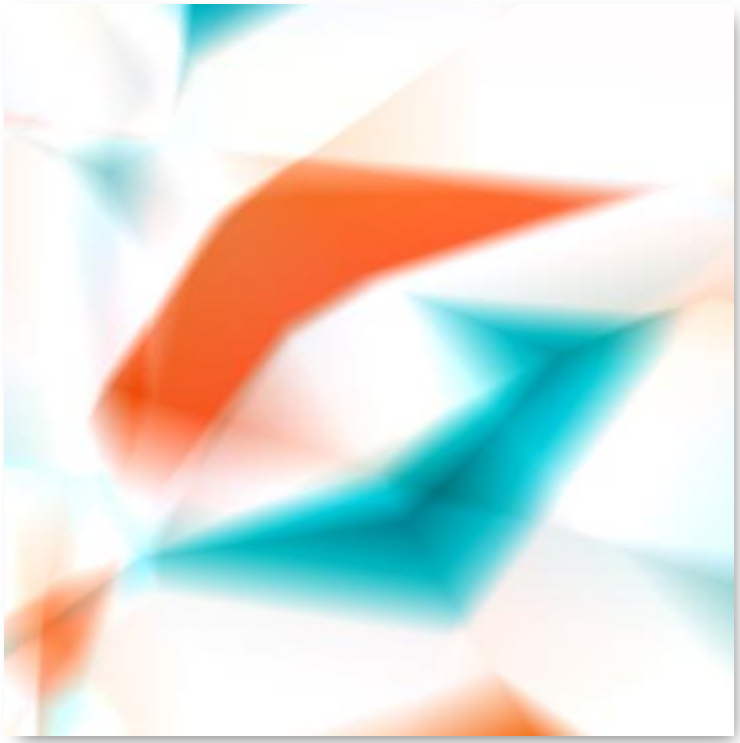
Reference

3 layers, 32 neurons

# ADDING MORE PARAMETERS?



Reference



3 layers, 32 neurons

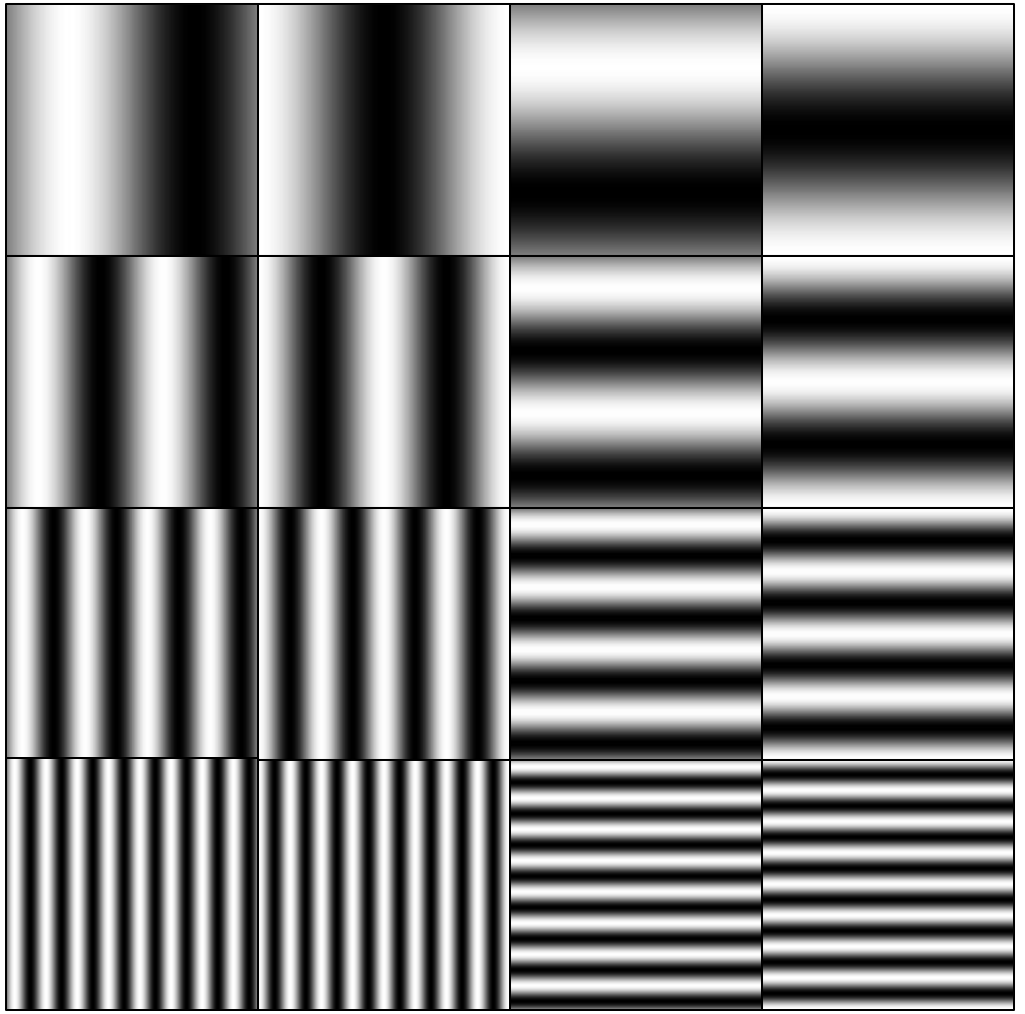
# GIVING BETTER INPUTS



“Frequency encoding”:

- Instead of passing u and v to the network…
- Pass

```
sin(u),  cos(u),  sin(v),  cos(v),
sin(2u), cos(2u), sin(2v), cos(2v),
sin(4u), cos(4u), sin(4v), cos(4v),
…
```

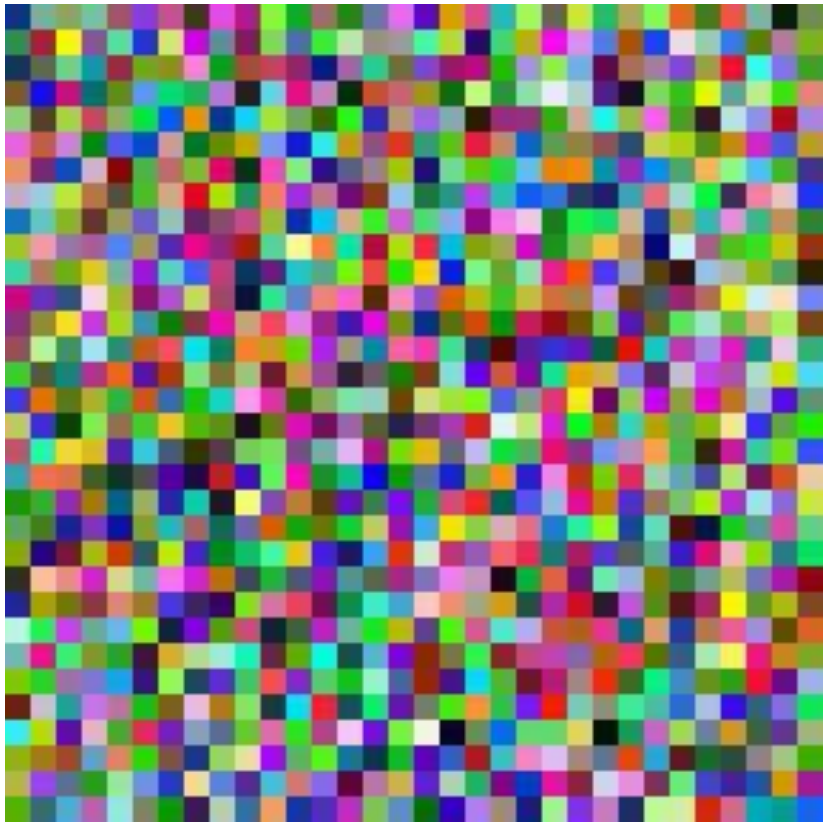


# LATENT TEXTURE





Neural Texture



Learned Latent Texture

# TAKE-AWAYS



- NOT: How to build a state-of-the-art neural texture compressor (that comes later in the course ☺)

# TAKE-AWAY #1



*Small* trainable models need (your!) careful engineering to work well.

E.g.

- Don’t make it learn things you already know.
  - Turn your domain knowledge into fixed components of your model
- Build up a tool box of common optimization tricks:
  - Different architectures, optimizers, activations, data preprocessing, ensembling, …
  - Learn how to debug when things don’t work

A lot of problems go away when your model has 8 billion parameters

- But that’s not us ☺

# TAKE-AWAY #2



Your code *has* to be fast to iterate

- Solving a new task with a trainable model will rarely work right away
- You might have to put on your *applied researcher* hat ☺
  - Experiment, debug, iterate

Anything that stands in the way of fast prototyping might stop you from succeeding

- Manual derivatives
- Boilerplate (pipeline setup, shader variable binding, …)

# TAKE-AWAY #2



Your code *has* to be fast to iterate

The Slang ecosystem is built to tackle these challenges (auto-diff, slangpy, …)

- Driving Slang from Python is very friendly to this kind of rapid exploration
- Once you have a model that works – ship it in C++
  - But you can reuse the Slang code!

SIGGRAPH 2025
Vancouver+ 10-14 August

# AUTOMATIC DIFFERENTIATION

# A DEEP DIVE INTO SLANG'S AUTODIFF FEATURE



**In this section:**

- Forward and backward gradient propagation
- How does auto-diff work with custom types
- Custom derivatives
- Propagate gradients to buffers

# CONSIDER A SIMPLE FUNCTION



```
float square(float x, float y)
{
    return x * x + y * y;
}

void main()
{
    float x = 3.0f;
    float y = 4.0f;
    float result = square(x, y);
    printf("Result: %f\n", result);
}
```

```
D:\test> slangi example.slang
Result: 25.000000
```

# MARKING FUNCTIONS AS [DIFFERENTIABLE] ALLOWS SLANG TO GENERATE DERIVATIVE FUNCTIONS



```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}
```

# COMPUTING FORWARD DERIVATIVES



```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}
```

$$f(x, y) = x^2 + y^2$$

$$f'(x, y) = 2x + 2y$$

$$f'(x, y, dx, dy) = 2xdx + 2ydy$$

# COMPUTING FORWARD DERIVATIVES



```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}

void main()
{
    let x = diffPair(3.0, 1.0); // x = 3, ∂x/∂θ = 1
    let y = diffPair(4.0, 1.0); // y = 4, ∂y/∂γ = 1
    let result = fwd_diff(square)(x, y);
    printf("dResult: %f\n", result.d);
}
```

```
D:\test> slangi example.slang
dResult: 14.000000
```

**Given:** $x = f(\theta) \qquad y = g(\gamma)$

$s = \text{square}(x, y)$

**If we know:**

"dx": $\frac{\partial x}{\partial \theta}$ and "dy": $\frac{\partial y}{\partial \gamma}$

**Compute:**

$$\frac{\partial s}{\partial \theta} + \frac{\partial s}{\partial \gamma}$$

i.e. $$\frac{\partial s}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial s}{\partial y}\frac{\partial y}{\partial \gamma}$$

$$\underset{3}{2x}\,\underset{1}{dx} + \underset{4}{2y}\,\underset{1}{dy} = 6+8 = 14$$

# COMPUTING FORWARD DERIVATIVES



```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}

void main()
{
    let x = diffPair(3.0, 1.0); // x = 3, ∂x/∂θ = 1
    let y = diffPair(4.0, 0.0); // y = 4, ∂y/∂γ = 0
    let result = fwd_diff(square)(x, y);
    printf("dResult: %f\n", result.d);
}
```

```
D:\test> slangi example.slang
dResult: 6.000000
```

**Given:** $x = f(\theta)$

$s = \text{square}(x, y)$

**How do we find out just** $\frac{\partial s}{\partial x}$ **?**

$$\frac{\partial s}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial s}{\partial y}\frac{\partial y}{\partial \gamma}\,0$$

$\frac{\partial x}{\partial \theta}$: 1

# COMPUTING FORWARD DERIVATIVES



```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}

void main()
{
    let x = diffPair(3.0, 0.0); // x = 3, ∂x/∂θ = 0
    let y = diffPair(4.0, 1.0); // y = 4, ∂y/∂γ = 1
    let result = fwd_diff(square)(x, y);
    printf("dResult: %f\n", result.d);
}
```

```
D:\test> slangi example.slang
dResult: 8.000000
```

**Given:** $x = f(\theta)$

$s = \text{square}(x, y)$

**How do we find out just** $\frac{\partial s}{\partial y}$ **?**

$$\frac{\partial s}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial s}{\partial y}\frac{\partial y}{\partial \gamma}$$

$\frac{\partial x}{\partial \theta}$ = 0, $\frac{\partial y}{\partial \gamma}$ = 1

# COMPUTING BACKWARD DERIVATIVES



```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}

void main()
{
    var x = diffPair(3.0, 0.0);
    var y = diffPair(4.0, 0.0);
    let dLdSquare = 1.0f;
    bwd_diff(square)(x, y, dLdSquare);
    printf("dL/dx: %f, dL/dy: %f\n", x.d, y.d);
}
```

```
D:\test> slangi example.slang
dL/dx: 6.000000, dL/dy: 8.000000
```

**Given:** $s = \mathrm{square}(x, y)$

$l = f(s)$

**If we know:**

$$\frac{\partial l}{\partial s}$$

**Compute:**

$$\frac{\partial l}{\partial x} \quad \text{and} \quad \frac{\partial l}{\partial y}$$

i.e. $\frac{\partial l}{\partial s}\frac{\partial s}{\partial x}$ and $\frac{\partial l}{\partial s}\frac{\partial s}{\partial y}$

$\frac{\partial l}{\partial s}$: 1 $\qquad$ $\frac{\partial l}{\partial s}$: 1

# USING CUSTOM TYPES WITH AUTO-DIFF



**What if `square` function takes in a user defined type?**

```
struct Point
{
    float x;
    float y;
}

[Differentiable]
float square(Point p)
{
    return p.x * p.x + p.y * p.y;
}
```

# DIFFERENTIABLE AND NON-DIFFERENTIABLE TYPES



```
[Differentiable]
float run(int op, float a, float b)
{
    if (op == 0)
        return a + b;
    else
        return a
}

void test()
{
    fwd_diff(run)( ? );
}
```

Non-differentiable argument is passed in as-is.

```
func fwd_diff(run)(op: int, a: DifferentialPair<float>,
b: DifferentialPair<float>) -> DifferentialPair<float>
```

Defined in test.slang(2)

# DIFFERENTIABLE AND NON-DIFFERENTIABLE TYPES



```
[Differentiable]
float run(int op, float a, float b)
{
    if (op == 0)
        return a + b;
    else
        return a
}

void test()
{
    fwd_diff(run)(???);
}
```

Differentiable argument is passed in as a pair of primal and gradient values.

```
func fwd_diff(run)(op: int, a: DifferentialPair<float>,
b: DifferentialPair<float>) -> DifferentialPair<float>
```

Defined in test.slang(2)

# IDIFFERENTIABLE INTERFACE



**Slang defines a built-in `IDifferentiable` interface to represent differentiable types.**

```
interface IDifferentiable
{
    associatedtype Differential : Idifferentiable
        where Differential.Differential == Differential;
    static Differential dzero();
    static Differential dadd(Differential, Differential);
    static Differential dmul<T : __BuiltinRealType>(T, Differential);
};
```

**By default, these builtin types are differentiable:**

`half`, `float`, `double`

vector and matrix of floating point types

`T[]` (if `T` is differentiable).

# USING CUSTOM TYPES WITH AUTO-DIFF



```
struct Point
{
    float x;
    float y;
}

[Differentiable]
float square(Point p)
{
    return p.x * p.x
}

void main()
{
    fwd_diff(square)(??)
}
```

`Point` is treated as non-differentiable by the type system.

```
func fwd_diff(square)(p: Point) ->
DifferentialPair<float>
```

Defined in test.slang(8)

# USING CUSTOM TYPES WITH AUTO-DIFF



```
struct Point : IDifferentiable
{
    float x;
    float y;
}

[Differentiable]
float square(Point p)
{
    return p.x * p.x
}

void main()
{
    fwd_diff(square)(???)
}
```

Mark the type as differentiable, and the compiler will auto-synthesize the implementation to IDifferentiable

```
func fwd_diff(square)(p: DifferentialPair<Point>) ->
DifferentialPair<float>

Defined in test.slang(8)
```

# DEFINING CUSTOM DERIVATIVES



## Consider the `square` function:

```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}
```

## Custom Forward Derivative:

```
[ForwardDerivativeOf(square)]
DifferentialPair<float> squareFwd(
    DifferentialPair<float> x,
    DifferentialPair<float> y)
{
    return diffPair(square(x.p, y.p),
                    2.0f * (x.p * x.d + y.p * y.d));

}
```

# DEFINING CUSTOM DERIVATIVES



## Consider the `square` function:

```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}
```

## Custom Backward Derivative:

```
[BackwardDerivativeOf(square)]
void squareBwd(
    inout DifferentialPair<float> x,
    inout DifferentialPair<float> y,
    float dOut)
{
    x = diffPair(x.p, 2.0f * x.p * dOut);
    y = diffPair(y.p, 2.0f * y.p * dOut);
}
```

# PROPAGATE GRADIENTS INTO BUFFERS



## Consider the `square` function:

```
RWStructuredBuffer<float> yBuffer;

[Differentiable]
float square(float x, int yIdx)
{
    let y = yBuffer[yIdx];
    return x * x +  y * y;
}

[numthreads(128, 1, 1)]
void computeMain(int tid
{
    var x = diffPair(3.0
    bwd_diff(square)(x, tid, 1.0f);
}
```

What if y is read from a buffer at yIdx?

How does gradient propagate to y?

```
func bwd_diff(square)(x:
InOut<DifferentialPair<float>>, yIdx: int,
resultGradient: float) -> void
```

Defined in test.slang(3)

# PROPAGATE GRADIENTS INTO BUFFERS



**Consider the `square` function:**

```
RWStructuredBuffer<float> yBuffer;

[Differentiable]
float square(float x, int yIdx)
{
    let y = getY(yIdx);
    return x * x +  y * y;
}

float getY(int yIdx) { return yBuffer[yIdx]; }
```

Step 1: wrap buffer access into a function.

# PROPAGATE GRADIENTS INTO BUFFERS



**Consider the `square` function:**

```
RWStructuredBuffer<float> yBuffer;

[Differentiable]
float square(float x, int yIdx)
{
    let y = getY(yIdx);
    return x * x +  y * y;
}

float getY(int yIdx) { return yBuffer[yIdx]; }
```

```
RWStructuredBuffer<Atomic<float>> yGradBuffer;

[BackwardDerivativeOf(getY)]
void getYBwd(int yIdx, float dOut)
{
    yGradBuffer[yIdx] += dOut;
}
```

Step 1: wrap buffer access into a function.

Step 2: provide custom derivative for `getY`.

# DEBUGGING WITH CUSTOM DERIVATIVES



```
[Differentiable]
float square(float x, float y)
{
    let x2 = x * x;
    let y2 = y * y;
    return x2 + y2;
}

[numthreads(128, 1, 1)]
void computeMain(int tid : SV_DispatchThreadID)
{
    var x = diffPair(3.0);
    var y = diffPair(4.0);
    // compute dL/dx and dL/dy
    bwd_diff(square)(x, y, 1.0f);
}
```

Want to check the gradient of y2 when debugging? How to `printf` it out?

# DEBUGGING WITH CUSTOM DERIVATIVES



```
[Differentiable]
float square(float x, float y)
{
    let x2 = x * x;
    let y2 = debugGrad(y * y);
    return x2 + y2;
}

[numthreads(128, 1, 1)]
void computeMain(int tid : SV_DispatchThreadID)
{
    var x = diffPair(3.0);
    var y = diffPair(4.0);
    // compute dL/dx and dL/dy
    bwd_diff(square)(x, y, 1.0f);
}
```

```
float debugGrad(float x) { return x; }

[BackwardDerivativeOf(debugGrad)]
void debugGradBwd(
    inout DifferentialPair<float> x, float dOut)
{
    printf("Gradient is %f\n", dOut);
    x = diffPair(x.p, dOut);
}
```

Backward custom derivatives can be viewed as a “callback” on propagated gradients.

# MORE RESROUCES ON AUTODIFF



**What we covered:**

- Definition of forward/backward derivative propagation functions
- Initiating forward or backward mode differentiation
- Custom differentiable types
- Custom Derivatives and their uses

**Checkout out Slang's documentation to learn more:**

https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/07-autodiff.html

# BREAK

SIGGRAPH 2025
Vancouver+ 10-14 August

9:00am – 10:45am

- Fundamentals
- Core Concepts
- Automatic Differentiation

10:45am – 12:15pm

- Hardware Acceleration
- Performance Tips
- Deployment
- Neural Textures
- Neural Materials

**Course Material:**

https://shader-slang.org/landing/siggraph-25

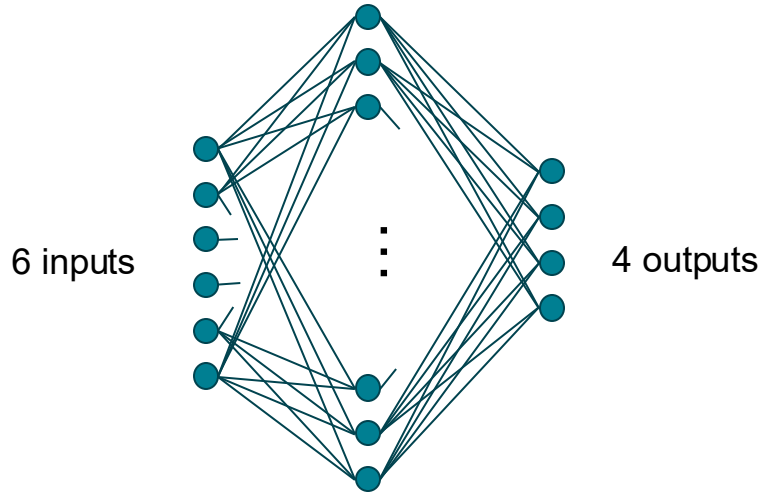
SIGGRAPH 2025
Vancouver+ 10-14 August

# HARDWARE ACCELERATION

# IMPLEMENTING OUR TOY NEURAL TEXTURE EXAMPLE



```
struct NeuralTexture2D
{
    NetworkParameters<6, 32> layer1;
    NetworkParameters<32, 4> layer2;

    float4 sample(float2 uv, float2 du, float2 dv)
    {
        float input[6] = {uv.x, uv.y, du.x, du.y, dv.x, dv.y};
        let latentResult = layer1.eval(input);
        let output = layer2.eval(latentResult);
        return float4(output[0], output[1], output[2], output[3]);
    }
}
```



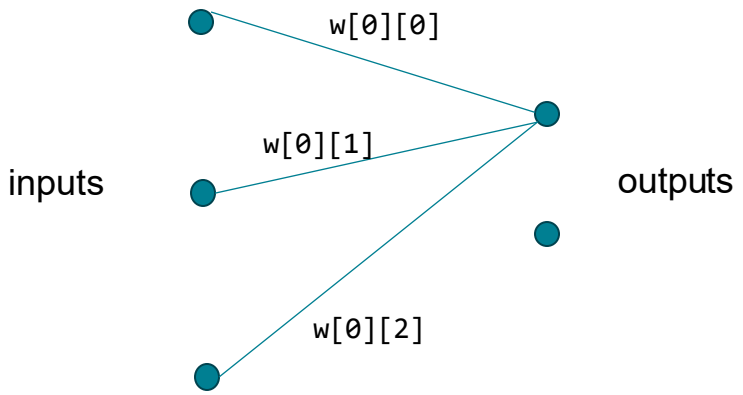
# COMPUTATION OF A 3-INPUT, 2-OUTPUT LAYER



```
outputs[0] = biases[0] +
             inputs[0] * w[0][0] +
             inputs[1] * w[0][1] +
             inputs[2] * w[0][2];
outputs[0] = max(0.0, outputs[0]);
```

Weighted Sum of Inputs

Activation



# COMPUTATION OF A 3-INPUT, 2-OUTPUT LAYER



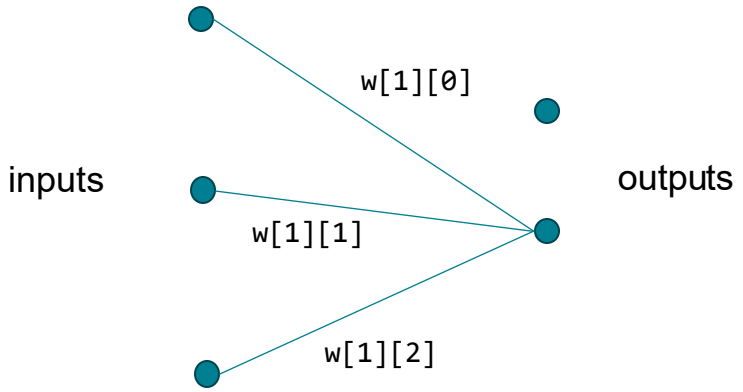
```
outputs[0] = biases[0] +
             inputs[0] * w[0][0] +
             inputs[1] * w[0][1] +
             inputs[2] * w[0][2];
outputs[0] = max(0.0, outputs[0]);

outputs[1] = biases[1] +
             inputs[0] * w[1][0] +
             inputs[1] * w[1][1] +
             inputs[2] * w[1][2];
outputs[1] = max(0.0, outputs[1]);
```

Weighted Sum of Inputs

Activation

Weighted Sum of Inputs

Activation

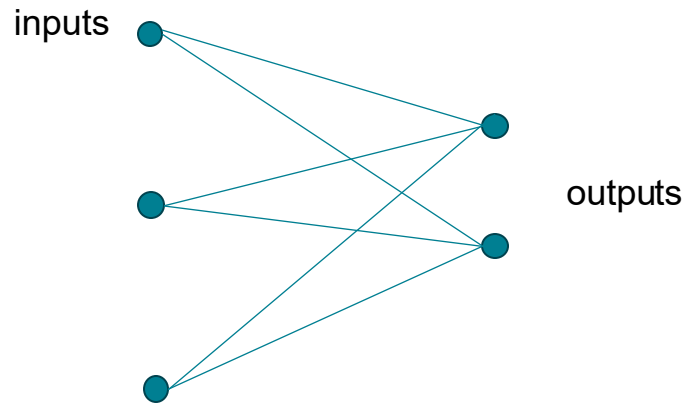


# COMPUTATION OF A 3-INPUT, 2-OUTPUT LAYER



```
outputs[0] = biases[0] +
             inputs[0] * w[0][0] +
             inputs[1] * w[0][1] +
             inputs[2] * w[0][2];
outputs[0] = max(0.0, outputs[0]);

outputs[1] = biases[1] +
             inputs[0] * w[1][0] +
             inputs[1] * w[1][1] +
             inputs[2] * w[1][2];
outputs[1] = max(0.0, outputs[1]);
```

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} + \begin{bmatrix} w_{00} & w_{01} & w_{02} \\ w_{10} & w_{11} & w_{12} \end{bmatrix} \begin{bmatrix} i_0 \\ i_1 \\ i_2 \end{bmatrix}$$



# SIMPLIFIED VIEW OF A FEED-FORWARD LAYER



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    float weights[InputSize][OutputSize];
    float biases[OutputSize];
    Array<float, OutputSize> eval(float input[InputSize])
    {
        float output[OutputSize] = biases;
        output += MatMulVec(weights, input);
        for (int i = 0; i < OutputSize; ++i)
            output[i] = max(output[i] * 0.01f, output[i]); // ReLU
        return output;
    }
}
```

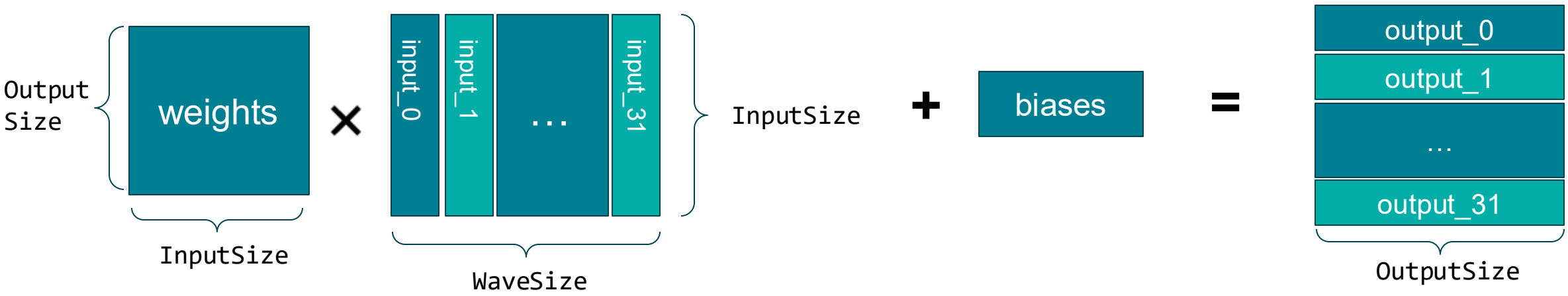
# EACH SUBGROUP IS COMPUTING A MATRIX-MATRIX MULTIPLICATION



```
Array<float, OutputSize> eval(float input[InputSize])
{
    float output[OutputSize] = biases;
    output += MatMulVec(weights, input);
    ...
    return output;
}
```

| Thread | Computation |
|---|---|
| 0 | MatMulVec(weights, input_0) |
| 1 | MatMulVec(weights, input_1) |
| 2 | MatMulVec(weights, input_2) |
| 3 | MatMulVec(weights, input_3) |
| … | |
| 31 | MatMulVec(weights, input_31) |

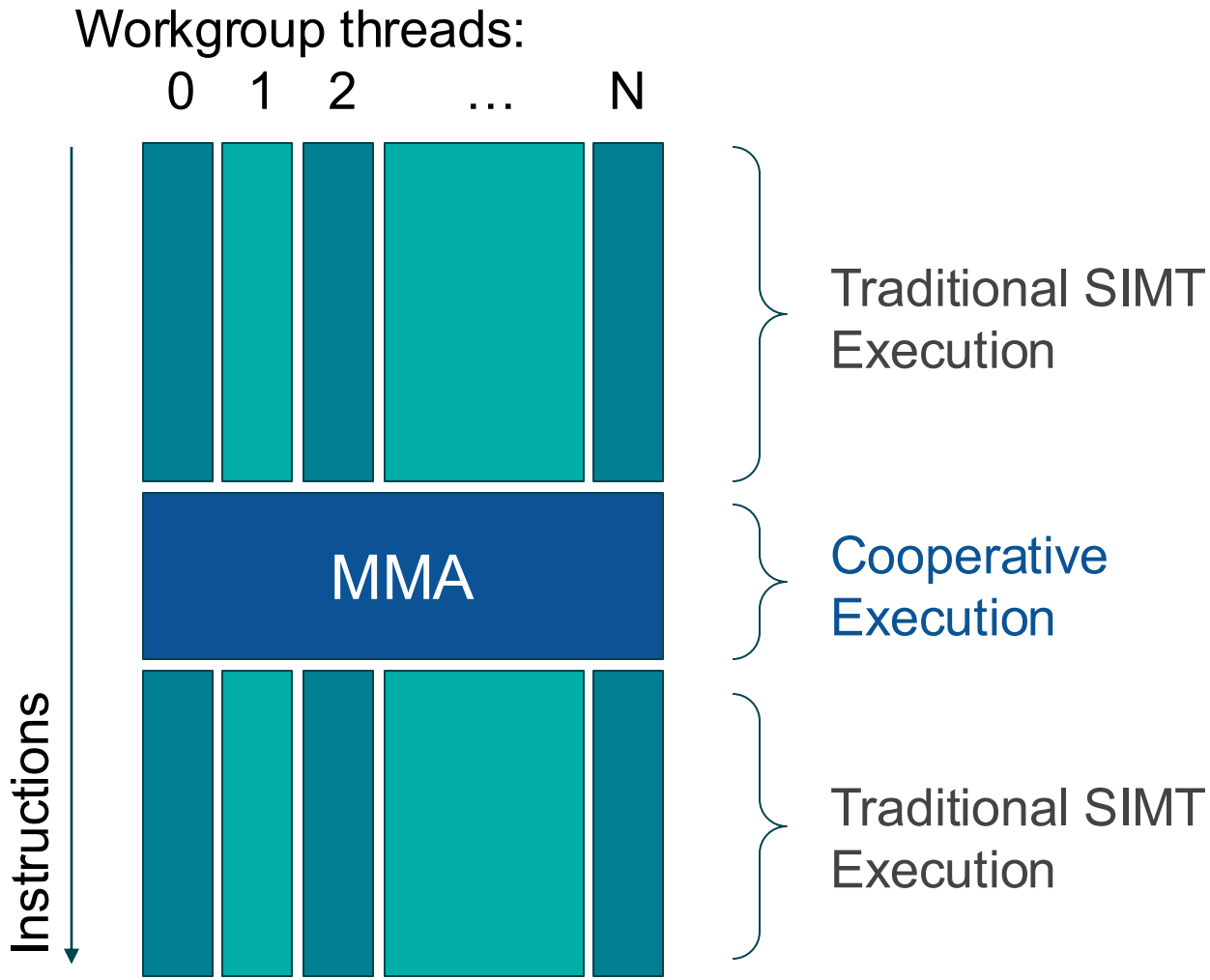
Computation done by the entire subgroup: **M**atrix **M**ultiply **A**ccumulate (a.k.a. **GE**neral **M**atrix **M**ultiply)




- *Wave == *Subgroup(Vulkan)* == *Warp* (CUDA) == *SIMD Group* (Metal)

# MODERN GPUS HAVE HARDWARE SUPPORT FOR MATRIX-MULTIPLY-ACCUMULATION





# MMA EXPOSED AS INTRINSICS IN VARIOUS APIS



DirectX® 12 — WaveMatrix

Vulkan™ — VK_KHR_cooperative_matrix

VK_NV_cooperative_matrix2

SIMD Group Matrix

# MMA INTRINSICS RESTRICTIONS



- Matrix operands in a Matrix-Multiply-Accumulation operation must be the same across all threads in a subgroup.

- Input vectors must be explicitly packed into a matrix and loaded in one operation.
  - Often require explicit use of group shared memory, which isn't available outside compute shaders.

# COOPERATIVE VECTORS: USING MMA HARDWARE FROM SIMT CODE



DirectX® 12

Shader Model 6.9

Vulkan™

VK_NV_cooperative_vector

Metal 4

Machine Learning

# SIMPLIFIED VIEW OF A FEED-FORWARD LAYER



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    float weights[InputSize][OutputSize];
    float biases[OutputSize];
    Array<float, OutputSize> eval(float input[InputSize])
    {
        float output[OutputSize] = biases;
        output += MatMulVec(weights, input);
        for (int i = 0; i < OutputSize; ++i)
            output[i] = max(output[i] * 0.01f, output[i]); // ReLU
        return output;
    }
}
```

# FEED-FORWARD LAYER USING COOPERATIVE VECTORS



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    half* weights;
    half* biases;
    CoopVec<half, OutputSize> eval(CoopVec<half, InputSize> input)
    {
        let output = coopVecMatMulAdd<half, OutputSize>(
            input, CoopVecComponentType.Float16, // input and format
            weights, CoopVecComponentType.Float16, // weights and format
            biases, CoopVecComponentType.Float16, // biases and format
            CoopVecMatrixLayout.RowMajor, // matrix layout
            false, // transpose matrix before multiply?
            sizeof(half) * InputSize); // matrix stride
        return max(output * 0.01h, output); // Leaky ReLU activation
    }
}
```

Using 16-bit floats because 32-bit float is not supported by HW.

# TRAINING WITH COOPERATIVE VECTORS



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    half* weights;
    half* biases;

    CoopVec<half, OutputSize> eval(CoopVec<half, InputSize> input)
    {
        let output = coopVecMatMulAdd<half, OutputSize>(
            input, CoopVecComponentType.Float16, // input and format
            weights, CoopVecComponentType.Float16, // weights and format
            biases, CoopVecComponentType.Float16, // biases and format
            CoopVecMatrixLayout.RowMajor, // matrix layout
            false, // transpose matrix before multiply?
            sizeof(half) * InputSize); // matrix stride
        return max(output * 0.01h, output); // Leaky ReLU activation
    }
}
```

1. Where should we store the gradients of the weights?

2. How to make this differentiable?

# STORING THE GRADIENTS OF THE WEIGHTS



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    half* weights;
    half* biases;
    half* weightsGrad;
    half* biasesGrad;

}
```

1. Where should we store the gradients of the weights?

# TRAINING WITH COOPERATIVE VECTORS



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    ...

    CoopVec<half, OutputSize> eval(CoopVec<half, InputSize> input)
    {
        let output = coopVecMatMulAdd<half, OutputSize>(
            input, CoopVecComponentType.Float16, // input and format
            weights, CoopVecComponentType.Float16, // weights and format
            biases, CoopVecComponentType.Float16, // biases and format
            CoopVecMatrixLayout.RowMajor, // matrix layout
            false, // transpose matrix before multiply?
            sizeof(half) * InputSize); // matrix stride
        return max(output * 0.01h, output); // Leaky ReLU activation
    }
}
```

How to make `eval` differentiable?

- `CoopVec` type is non-differentiable.
- `coopVecMatMulAdd` intrinsic is non-differentiable.

# WRAPPING COOPERATIVE VECTOR IN A DIFFERENTIABLE TYPE



```
struct MLVec<int N> : IDifferentiable
{
    CoopVec<half, N> data;
    typealias Differential = This;

    static Differential dadd(Differential d0, Differential d1)
    {
        return {d0.data + d1.data};
    }
    static Differential dmul<U:__BuiltinRealType>(U s, Differential d)
    {
        return {d.data * __realCast<half>(s)};
    }
    static Differential dzero()
    {
        return {};
    }

}
```

Wrap CoopVec in a MLVec type, that is declared to be Differentiable, with Differential type being itself.

# UPDATE EVAL TO USE MLVEC TYPE



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    ...
    half* weightsGrad;
    half* biasesGrad;

    MLVec<OutputSize> eval(MLVec<InputSize> input) { ... }


}
```

# MAKE EVAL DIFFERENTIABLE WITH CUSTOM DERIVATIVE



```
struct NetworkParameters<int InputSize, int OutputSize>
{
    ...
    half* weightsGrad;
    half* biasesGrad;

    MLVec<OutputSize> eval(MLVec<InputSize> input) { ... }

    [BackwardDerivativeOf(eval)]
    void evalBwd(
        inout DifferentiablePair<MLVec<InputSize> input,
        MLVec<OutputSize> resultGrad)
    {
        // What goes here??
    }
}
```

**Task: propagate gradients from each output element (`resultGrad`) to:**

- weight and bias parameters (to store in `weightsGrad` and `biasesGrad`).
- Each input element (return via `input.d`)

# GRADIENT PROPAGATION FOR MATRIX-VECTOR MULTIPLICATION



Given matrix-vector multiplication

$$\boldsymbol{u} = \mathrm{M}\boldsymbol{v} + B$$

We have:

$$d\mathrm{M} = d\mathbf{u} \otimes_{\text{outer}} \boldsymbol{v}$$

$$d\mathrm{B} = d\mathbf{u}$$

$$d\mathbf{v} = \mathrm{M}^T\, d\mathbf{u}$$

$\otimes_{\text{outer}}$ : outer vector product

# ACCUMULATING GRADIENTS ACROSS THREADS



Given matrix-vector multiplication

$$\boldsymbol{u} = \mathrm{M}\boldsymbol{v} + B$$

We have:

$$d\mathrm{M} = d\mathbf{u} \otimes_{\text{outer}} \boldsymbol{v}$$

$$d\mathrm{B} = d\mathbf{u}$$

$$d\mathbf{v} = \mathrm{M}^T\, d\mathbf{u}$$

$\otimes_{\text{outer}}$ : outer vector product

M

dM

B

dB

| Thread | Computation |
|---|---|
| 0 | u = malMulVecAdd(M, v0, B) |
| 1 | u = malMulVecAdd(M, v1, B) |
| 2 | u = malMulVecAdd(M, v2, B) |
| … | |
| N | u = malMulVecAdd(M, vN, B) |

Each thread accesses different v, but the same M and B

# ACCUMULATING GRADIENTS ACROSS THREADS



Given matrix-vector multiplication

$$\boldsymbol{u} = \mathrm{M}\boldsymbol{v} + B$$

We have:

$$d\mathrm{M} \mathrel{+}= d\mathbf{u} \otimes_{\text{outer}} \boldsymbol{v}$$

$$d\mathrm{B} \mathrel{+}= d\mathbf{u}$$

$$d\mathbf{v} = \mathrm{M}^T\, d\mathbf{u}$$


| Thread | Computation |
|---|---|
| 0 | u = malMulVecAdd(M, v0, B)<br>dM += outerProduct(…)<br>dB += dU |
| 1 | u = malMulVecAdd(M, v1, B)<br>dM += outerProduct(…)<br>dB += dU |
| 2 | u = malMulVecAdd(M, v2, B)<br>dM += outerProduct(…)<br>dB += dU |
| … | |
| N | u = malMulVecAdd(M, vN, B)<br>dM += outerProduct(…)<br>dB += dU |

Needs to be atomic to prevent race conditions!

# INTRINSICS TO SPEEDUP ATOMIC GRADIENT ACCUMULATION



Given matrix-vector multiplication

$$\boldsymbol{u} = \mathrm{M}\boldsymbol{v} + B$$

We have:

$$d\mathrm{M} \mathrel{+}= d\mathbf{u} \otimes_{\text{outer}} \boldsymbol{v}$$ ← `coopVecOuterProductAccumulate`

$$d\mathrm{B} \mathrel{+}= d\mathbf{u}$$ ← `coopVecReduceSumAccumulate`

$$d\mathbf{v} = \mathrm{M}^T\, d\mathbf{u}$$ ← `coopVecMatMul`

Each step has its corresponding coop-vector intrinsic:

# BACK-PROP GRADIENTS THROUGH EVAL



```
[BackwardDerivativeOf(eval)]
void evalBwd(
    inout DifferentialPair<MLVec<InputSize>> input,
    MLVec<OutputSize> resultGrad)
{
    let fwd = eval(input.p);
    // Back-prop resultGrad through activation.
    for (int i = 0; i < OutputSize; i++)
        if (fwd.data[i] < 0.0)
            resultGrad.data[i] *= 0.01h;

    // Back-prop gradients to the weights matrix.
    coopVecOuterProductAccumulate(
        resultGrad.data,
        input.v.data,
        weightsGrad, 0,
        CoopVecMatrixLayout.TrainingOptimal, CoopVecComponentType.Float16);

    // Back-prop gradients to the biases vector.
    coopVecReduceSumAccumulate(resultGrad.data, biasesGrad, biasesOffset);
```

`coopVecOuterProductAccumulate` **requires** `weightsGrad` to be stored in `TrainingOptimal` layout.

$$d\mathrm{M} \mathrel{+}= d\mathbf{u} \otimes_{\mathrm{outer}} \boldsymbol{v}$$

$$d\mathrm{B} \mathrel{+}= d\mathbf{u}$$

# BACK-PROP GRADIENTS THROUGH EVAL



```
[BackwardDerivativeOf(eval)]
void evalBwd(
    inout DifferentialPair<MLVec<InputSize>> input,
    MLVec<OutputSize> resultGrad)
{
    ...
    // Back-prop gradients to the input vector.
    let dInput = coopVecMatMul<half, InputSize>(
        resultGrad.data, CoopVecComponentType.Float16,
        weights, CoopVecComponentType.Float16,
        CoopVecMatrixLayout.ColumnMajor, false, sizeof(half)*InputSize);
    input = {input.p, {dInput}};
}
```

$$d\mathbf{v} = \mathrm{M}^T \, d\mathbf{u}$$

Compute
i.e. `transpose(weights) * resultGrad`
by setting the layout of weights to **`ColumnMajor`**.

# CALLERS OF FEED-FORWARD LAYER CAN NOW BE AUTODIFF'ED



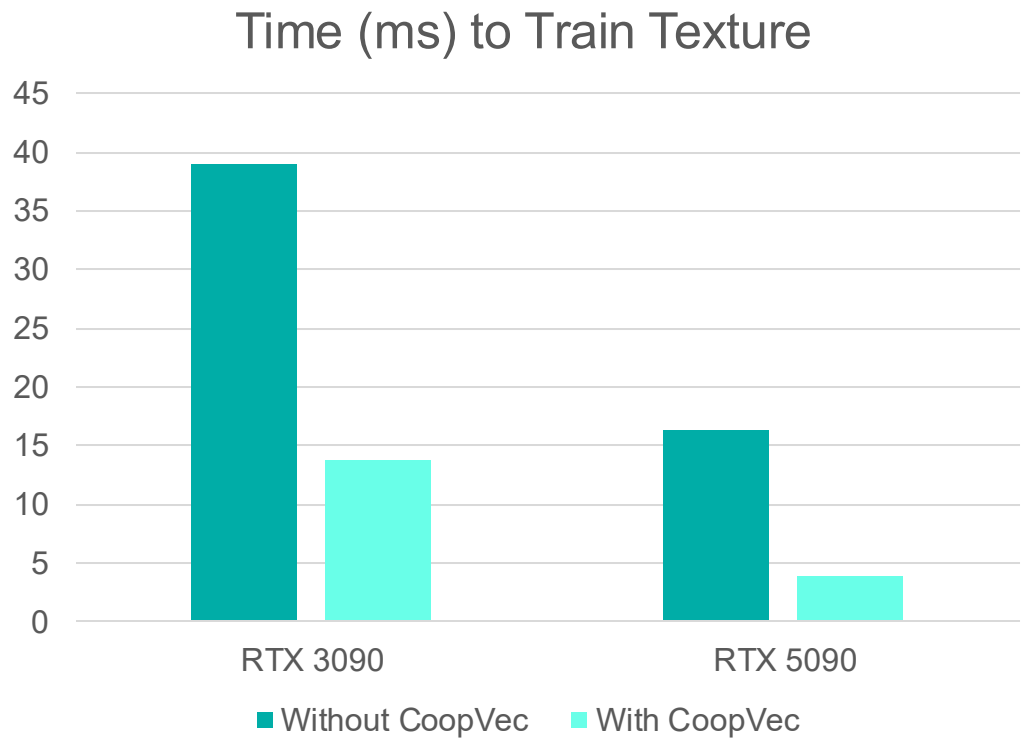
```
struct NeuralTexture2D
{
    NetworkParameters<6, 32> layer1;
    NetworkParameters<32, 4> layer2;

    [Differentiable]
    float4 sample(float2 uv, float2 du, float2 dv)
    {
        float input[6] = {uv.x, uv.y, du.x, du.y, dv.x, dv.y};
        let latentResult = layer1.eval(input);
        let output = layer2.eval(latentResult);
        return float4(output[0], output[1], output[2], output[3]);
    }
}
```

# PERFORMANCE IMPROVEMENTS WITH COOP-VECTOR



- **Benchmark:** Neural Texture Training
- **Input:** **100MB** PNG

Over **3x** speedup compared to highly optimized code without coop-vec.



# RESOURCES



- **Source code and other resources:**
  - Course page: http://bit.ly/4okm8Vm
  - Checkout the “MLP-Training-CoopVec” example for both shader (Slang) and host-side (C++) code.
  - NVIDIA Neural Shading SDK: https://github.com/NVIDIA-RTX/RTXNS

**Course Landing Page**

http://bit.ly/4okm8Vm

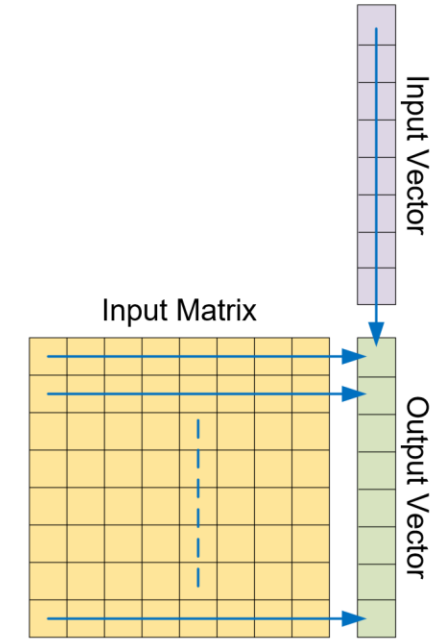
SIGGRAPH 2025
Vancouver+ 10-14 August

# PERFORMANCE TIPS

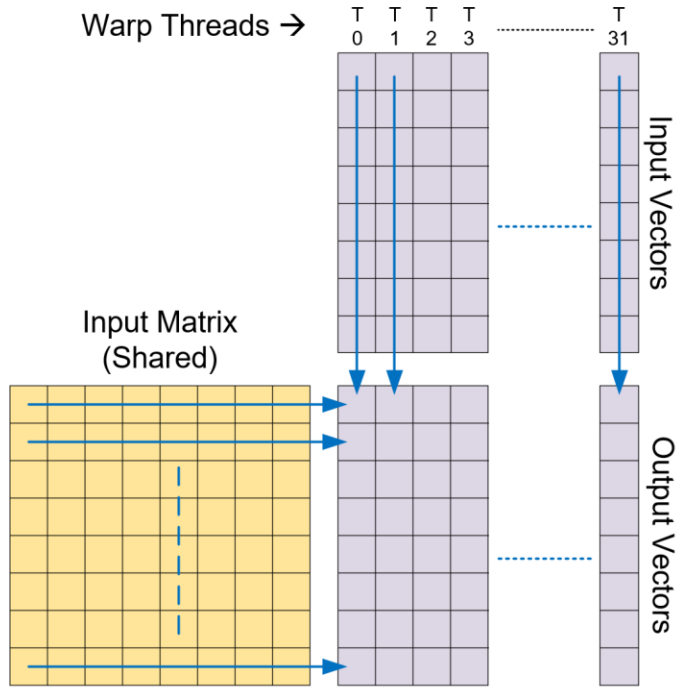
# BATCHED MATRIX-VECTOR MULTIPLICATION



- Hardware Tensor Cores:
  - Matrix-Matrix multiplication using entire wave/warp
  - Low precision (FP16, FP8, INT8, even FP4)
- Cooperative Vector API:
  - Matrix-Vector multiplication in each thread
- Mapping CoopVec onto Tensor Cores:
  - Combine M-V multiplications from all threads in a wave
  - Divergence becomes a problem (more on that later)




Single Thread:
Matrix-Vector Multiplication



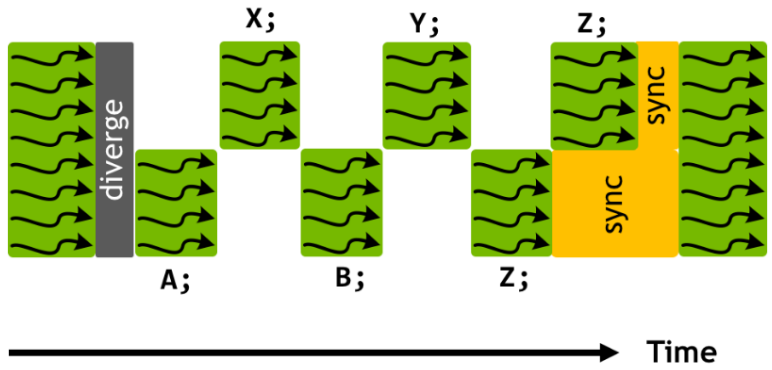

Many Threads:
Matrix-Matrix Multiplication
Maps to Tensor Cores

# REFRESHER ON GPU SIMT ARCHITECTURE



- NVIDIA GPUs group execution threads into 32-thread “warps”
  - Other vendors have similar concepts with different thread counts
  - We use the term “wave” throughout this course for consistency
- All threads in a wave execute one instruction on different data
  - Some may be inactive
  - Following different code branches in a wave causes execution divergence
  - Divergent code branches are serialized into synchronous groups
- Memory accesses work best when addresses are packed and aligned
  - Not following optimal address patterns causes data divergence
  - Divergent memory accesses are serialized into optimal groups

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

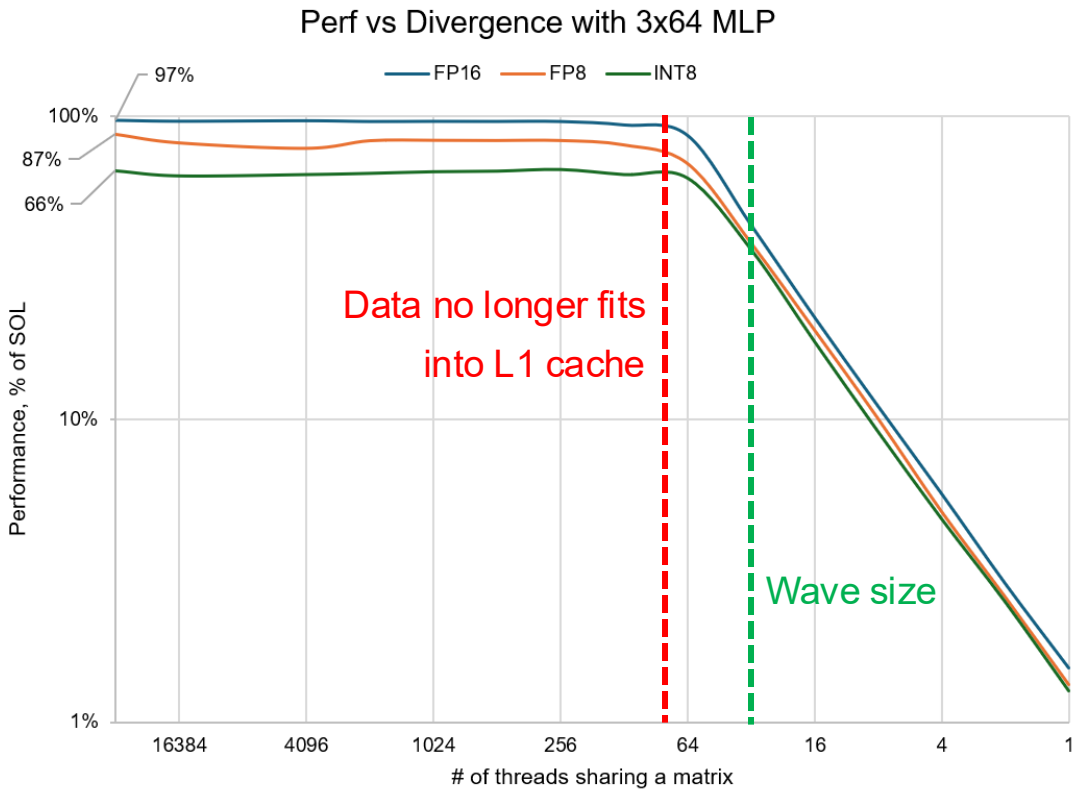



*Source: Volta architecture whitepaper*

# TENSOR CORES AND DIVERGENCE



- Tensor Cores compute a single MMA using the entire wave
  - MMA = Matrix Multiply Accumulate
  - We can wake up inactive threads if needed
- Wave-wide coopVecMatMul[Add] might be incompatible:
  - Vector inputs are trivially combined into a matrix
  - Matrix inputs must be the same for all threads
  - Programming model allows matrix inputs to be different per-thread
- Solution: Serialize divergent matrix operations
  - Handled transparently by the driver
  - Has severe performance effects
- Avoid CoopVec operations with divergent matrices
  - Group draw calls by material
  - Sort threads manually or use Shader Execution Reordering (SER)




Measured on RTX 4090 using a directed test – Log/Log scale

https://github.com/jeffbolznv/vk_cooperative_vector_perf

# TENSOR CORE MATRIX LAYOUT PROBLEM



- Tensor cores use custom matrix layouts
  - Components of matrices are shuffled between threads
  - Specific layout depends on the GPU and data types
- Need to shuffle data before and after MMAs
  - Weight matrix must be pre-shuffled in memory
  - DX12 and Vulkan provide functions to shuffle the weights
  - Input and output shuffling is handled transparently by the driver
- Output of one MMA can be used as input of another MMA without shuffle
  - Redundant shuffles can be eliminated

MMA computation 1

| Row\ Col | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | T0 : { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 3 | T3: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| 4 | T16: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 7 | T19: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |

MMA computation 3

| Row\Col | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | T8 : { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 3 | T11: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| 4 | T24: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 7 | T27: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |

MMA computation 2

| Row\Col | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | T4 : { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 3 | T7: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| 4 | T20: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 7 | T23: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |

MMA computation 4

| Row\Col | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | T12 : { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 3 | T15: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| 4 | T28: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |
| .. | | | | | | | | |
| 7 | T31: { c0, c1, c2, c3, c4, c5, c6, c7 } | | | | | | | |

Output matrix layout for an FP16 MMA operation (maps to 4 instructions)
https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

# WEIGHT MATRIX LAYOUT CONVERSIONS



- Query shuffled matrix size from VK/DX12
  - Could be larger than the optimally packed data
- Upload row- or column-major matrix to GPU
- Convert layout (perform the shuffle) on the GPU
  - Inference Optimal
  - Training Optimal
- Use converted matrix in CoopVec operations
  - Specify layout as `coopVecMatMulAdd` argument

## FEED-FORWARD LAYER USING COOPERATIVE VECTORS

```
struct NetworkParameters<int InputSize, int OutputSize>
{
    half* weights;
    half* biases;
    CoopVec<half, OutputSize> eval(CoopVec<half, InputSize> input)
    {
        let output = coopVecMatMulAdd<half, OutputSize>(
            input, CoopVecComponentType.Float16,
            weights, CoopVecComponentType.Float16,
            biases, CoopVecComponentType.Float16,
            CoopVecMatrixLayout.InferencingOptimal,
            false, // is matrix transposed
            sizeof(half) * InputSize); // matrix stride
        return max(output * 0.01h, output);
    }
}
```

PREVIOUS SECTION

# MATRIX LAYOUT CONVERSION APIS



## Vulkan (VK_NV_cooperative_vector)

- `vkConvertCooperativeVectorMatrixNV`
  - CPU conversion / size query
- `vkCmdConvertCooperativeVectorMatrixNV`
  - GPU conversion

## DX12 (Agility SDK 717 Preview)

- `Device::GetLinearAlgebraMatrixConversion…DestinationInfo`
  - Size query
- `CommandList::ConvertLinearAlgebraMatrix`
  - GPU conversion
  - Note: No CPU conversion function!

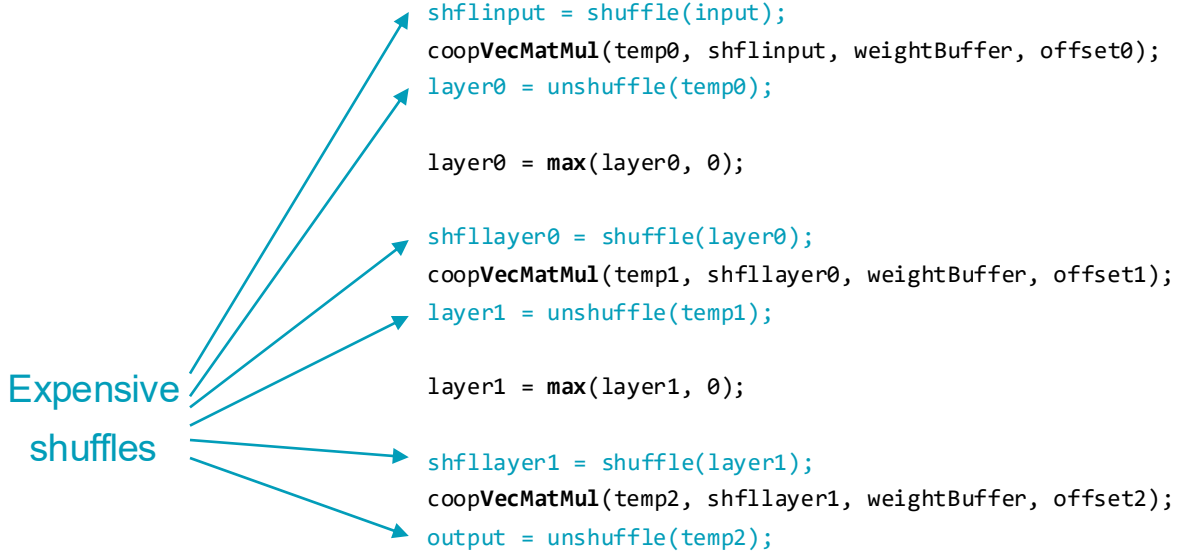
# MAPPING AN MLP ONTO TENSOR CORES 1/3



## ORIGINAL CODE

```
coopVecMatMul(layer0, input, weightBuffer, offset0);
layer0 = max(layer0, 0);

coopVecMatMul(layer1, layer0, weightBuffer, offset1);
layer1 = max(layer1, 0);

coopVecMatMul(output, layer1, weightBuffer, offset2);
```

## ADD SHUFFLES AROUND MMA OPS




```
shflinput = shuffle(input);
coopVecMatMul(temp0, shflinput, weightBuffer, offset0);
layer0 = unshuffle(temp0);

layer0 = max(layer0, 0);

shfllayer0 = shuffle(layer0);
coopVecMatMul(temp1, shfllayer0, weightBuffer, offset1);
layer1 = unshuffle(temp1);

layer1 = max(layer1, 0);

shfllayer1 = shuffle(layer1);
coopVecMatMul(temp2, shfllayer1, weightBuffer, offset2);
output = unshuffle(temp2);
```

*Baseline implementation – functional but slow*

# MAPPING AN MLP ONTO TENSOR CORES 2/3



```
shflinput = shuffle(input);
coopVecMatMul(temp0, shflinput, weightBuffer, offset0);
layer0 = unshuffle(temp0);

layer0 = max(layer0, 0);

shfllayer0 = shuffle(layer0);
coopVecMatMul(temp1, shfllayer0, weightBuffer, offset1);
layer1 = unshuffle(temp1);

layer1 = max(layer1, 0);

shfllayer1 = shuffle(layer1);
coopVecMatMul(temp2, shfllayer1, weightBuffer, offset2);
output = unshuffle(temp2);
```

## REMOVE UNSHUFFLE / SHUFFLE PAIRS

```
shflinput = shuffle(input);
coopVecMatMul(temp0, shflinput, weightBuff, offset0);
temp0_1 = max(temp0, 0);

coopVecMatMul(temp1, temp0_1, weightBuff, offset1);
temp1_1 = max(temp1, 0);

coopVecMatMul(temp2, temp1_1, weightBuff, offset1);
output = unshuffle(temp2);
```

*The shuffle removal is called “Layer fusion”*

*Sometimes it fails*

# MAPPING AN MLP ONTO TENSOR CORES 3/3



```
shflinput = shuffle(input);
coopVecMatMul(shfllayer0, shflinput, weightBuff, offset0);
temp0_1 = max(temp0, 0);

coopVecMatMul(temp1, temp0_1, weightBuff, offset1);
temp1_1 = max(temp1, 0);

coopVecMatMul(temp2, temp1_1, weightBuff, offset1);
output = unshuffle(temp2);
```

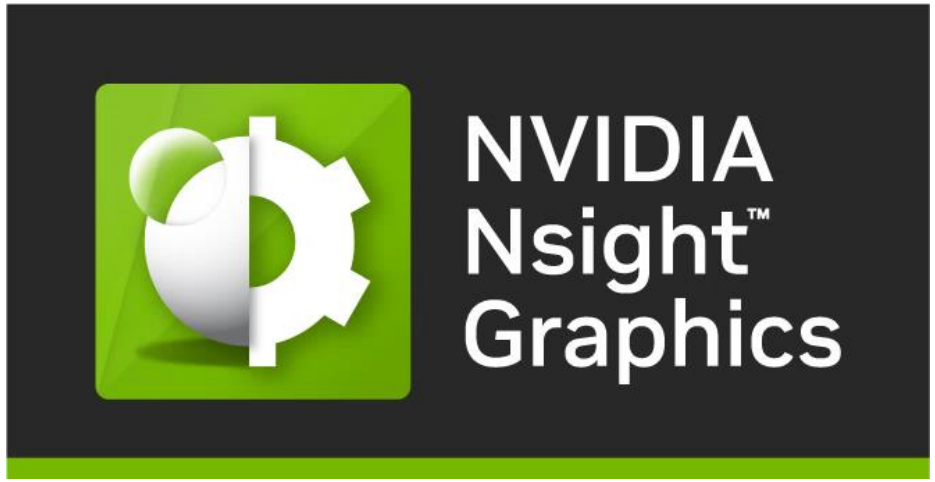
## PEEL THE DIVERGENT MATRICES

```
shflinput = shuffle(input);

foreach (unique combination of offsets)
{
    coopVecMatMul(temp0, shflinput, weightBuff, offset0);
    temp0_1 = max(temp0, 0);

    coopVecMatMul(temp1, temp0_1, weightBuff, offset1);
    temp1_1 = max(temp1, 0);

    coopVecMatMul(temp2, temp1_1, weightBuff, offset2);

    mergeThreadResults(temp2);
}

output = unshuffle(temp2);
```

# DIAGNOSING LAYER FUSION FAILURES



- No publicly available diagnostic tool at this time
- Experimental approach:
  - Measure performance of the original code
  - Remove all code between coopVecMatMul[Add] operations
  - See if performance increases dramatically
  - Re-introduce pieces of that code looking for perf cliffs




Nsight Graphics is a great tool,

but it doesn’t fully support CoopVec yet

# PREVENTING LAYER FUSION FAILURES 1/3



Avoid elementwise operations on CoopVec's between layers

## BAD

```
coopVecMatMul(layer0, input, weightBuffer, offset0);

for (int i = 0; i < 64; ++i)
{
    // Leaky ReLU
    if (layer0[i] < 0)
        layer0[i] *= 0.01;
}

coopVecMatMul(layer1, layer0, weightBuffer, offset1);
```

## GOOD

```
coopVecMatMul(layer0, input, weightBuffer, offset0);

// Use vector operations to express the same math
layer0 = max(layer0, layer0 * 0.01);

coopVecMatMul(layer1, layer0, weightBuffer, offset1);
```

Useful vector intrinsics: min, max, clamp, **step**

Look in "hlsl.meta.slang" for more

# PREVENTING LAYER FUSION FAILURES 2/3



Use vector load and store operations instead of elementwise ones

## BAD

```
coopVecMatMul(layer0, input, weightBuffer, offset0);

CoopVec<half, 64> bias;
for (int i = 0; i < 64; ++i)
{
    bias[i] = biasBuffer.Load<half>(biasOffset0
        + i * sizeof(half));
}

layer0 += bias;

layer0 = max(layer0, 0); // ReLU

coopVecMatMul(layer1, layer0, weightBuffer, offset1);
```

## BETTER

```
coopVecMatMul(layer0, input, weightBuffer, offset0);

let bias = CoopVec<half, 64>.load(biasBuffer, biasOffset0);
layer0 += bias;

layer0 = max(layer0, 0); // ReLU

coopVecMatMul(layer1, layer0, weightBuffer, offset1);
```

# PREVENTING LAYER FUSION FAILURES 3/3



Use vector load and store operations instead of elementwise ones

## BETTER

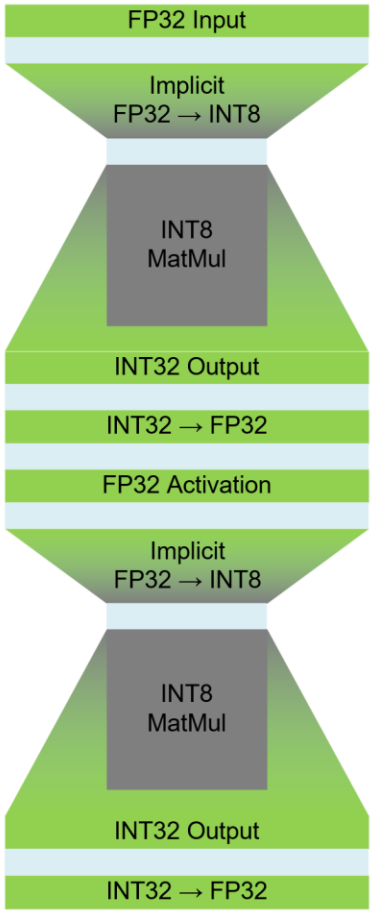
```
coopVecMatMul(layer0, input, weightBuffer, offset0);

let bias = CoopVec<half, 64>.load(biasBuffer, biasOffset0);
layer0 += bias;

layer0 = max(layer0, 0); // ReLU

coopVecMatMul(layer1, layer0, weightBuffer, offset1);
```

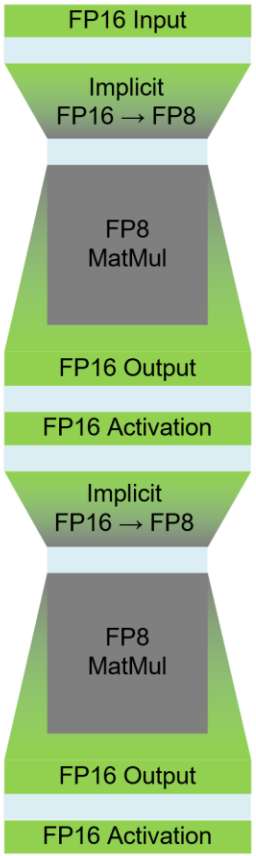
## GOOD

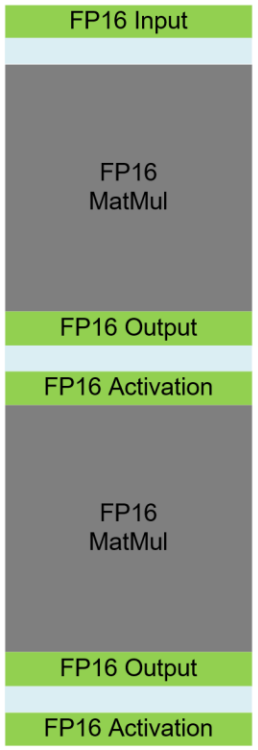
```
coopVecMatMulAdd(layer0, input, weightBuffer, offset0,
    biasBuffer, biasOffset0);

layer0 = max(layer0, 0); // ReLU

coopVecMatMul(layer1, layer0, weightBuffer, offset1);
```

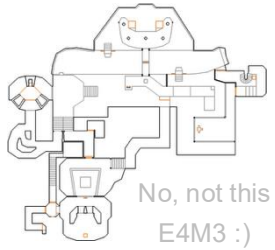
# MATH PRECISION CONSIDERATIONS





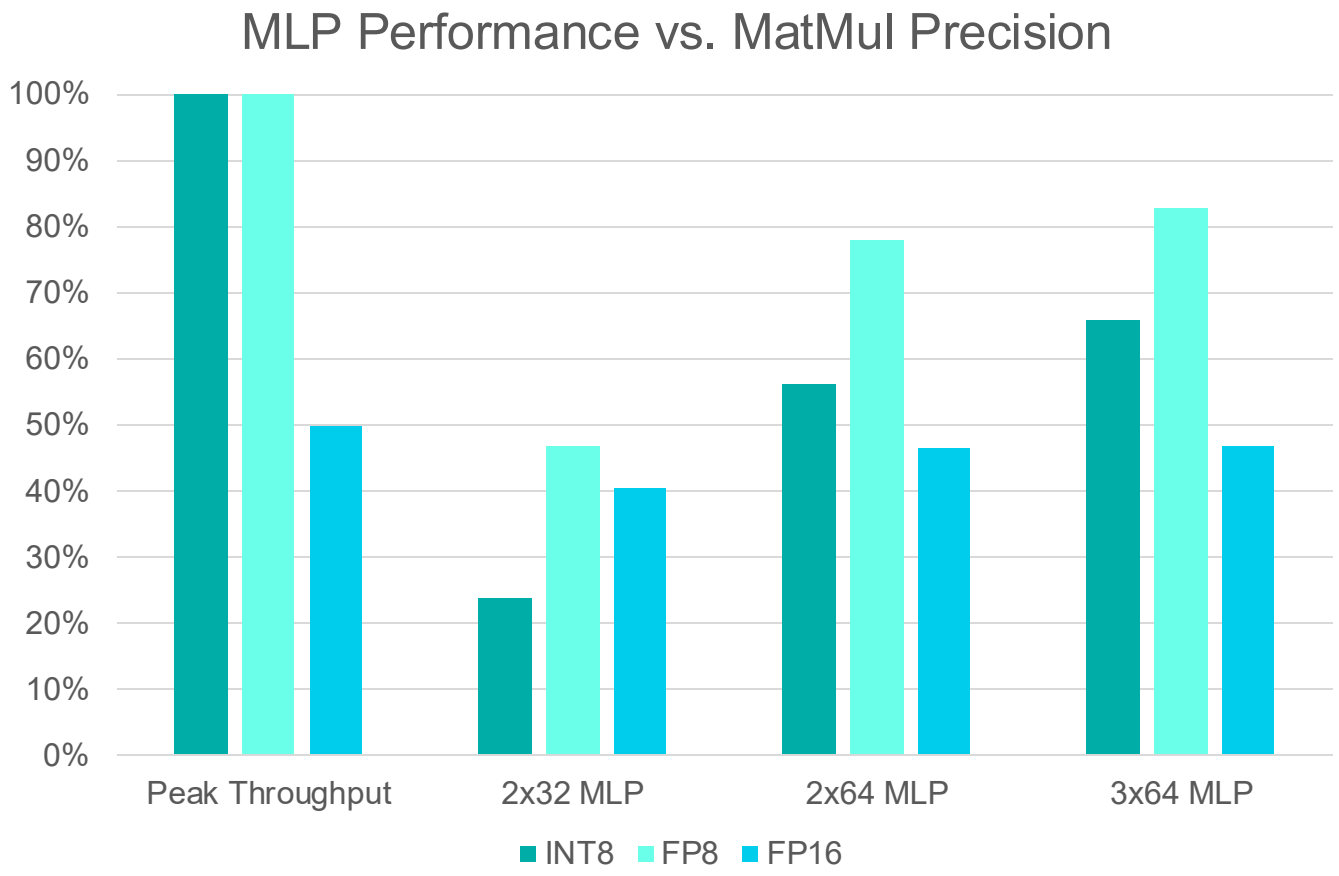




- INT8 – Most difficult to use
  - Extra scale operations needed for each layer
  - INT->FP conversions are not cheap
  - 32-bit per element vectors take many registers
- FP8 (E4M3 and E5M2)
  - No scaling or conversions needed, small register footprint
  - Very low precision, mostly suitable for hidden layers
- FP16 – Easiest to use
  - Lower peak throughput than FP8 or INT8
  - Enough precision for all practical inference needs

# MATH PRECISION COMPARISON





# PERFORMANCE TIPS - SUMMARY



- Convert matrices to optimal layouts offline
- Minimize matrix divergence
- Avoid elementwise access to vectors between network layers
- Watch out for layer fusion failures

Math Precision Tradeoffs

| | INT8 | FP8 | FP16 |
|---|---|---|---|
| Precision | ✓ | ✗ | ✓✓ |
| Throughput | ✓ | ✓✓ | ✗ |
| Register Pressure | ✗ | ✓ | ✓ |
| Ease of Use | ✗ | ✓ | ✓✓ |
| GPU Compatibility | ✓✓ | ✓ | ✓✓ |

SIGGRAPH 2025
Vancouver+ 10-14 August

# DEPLOYMENT

# THE TRANSITION STORY



**Training + Inference**

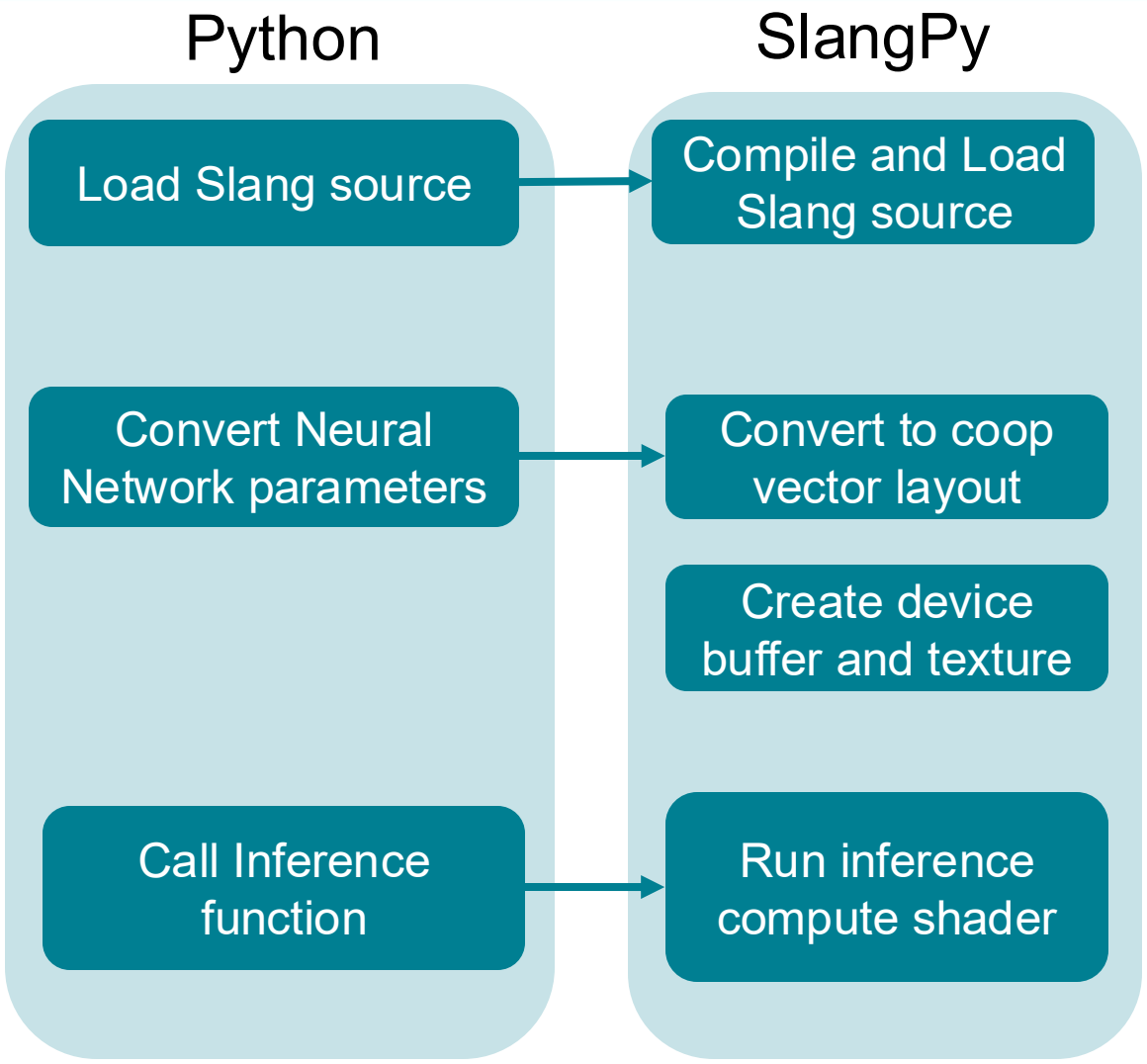

**Inference**


**SlangPy**



# PYTHON IMPLEMENTATION






```python
import slangpy as spy
module = spy.Module.load_from_file(app.device, "inference.slang")
app = App(width=512*3+10*2, height=512, title="Mipmap Example",
          device_type=spy.DeviceType.vulkan)
```

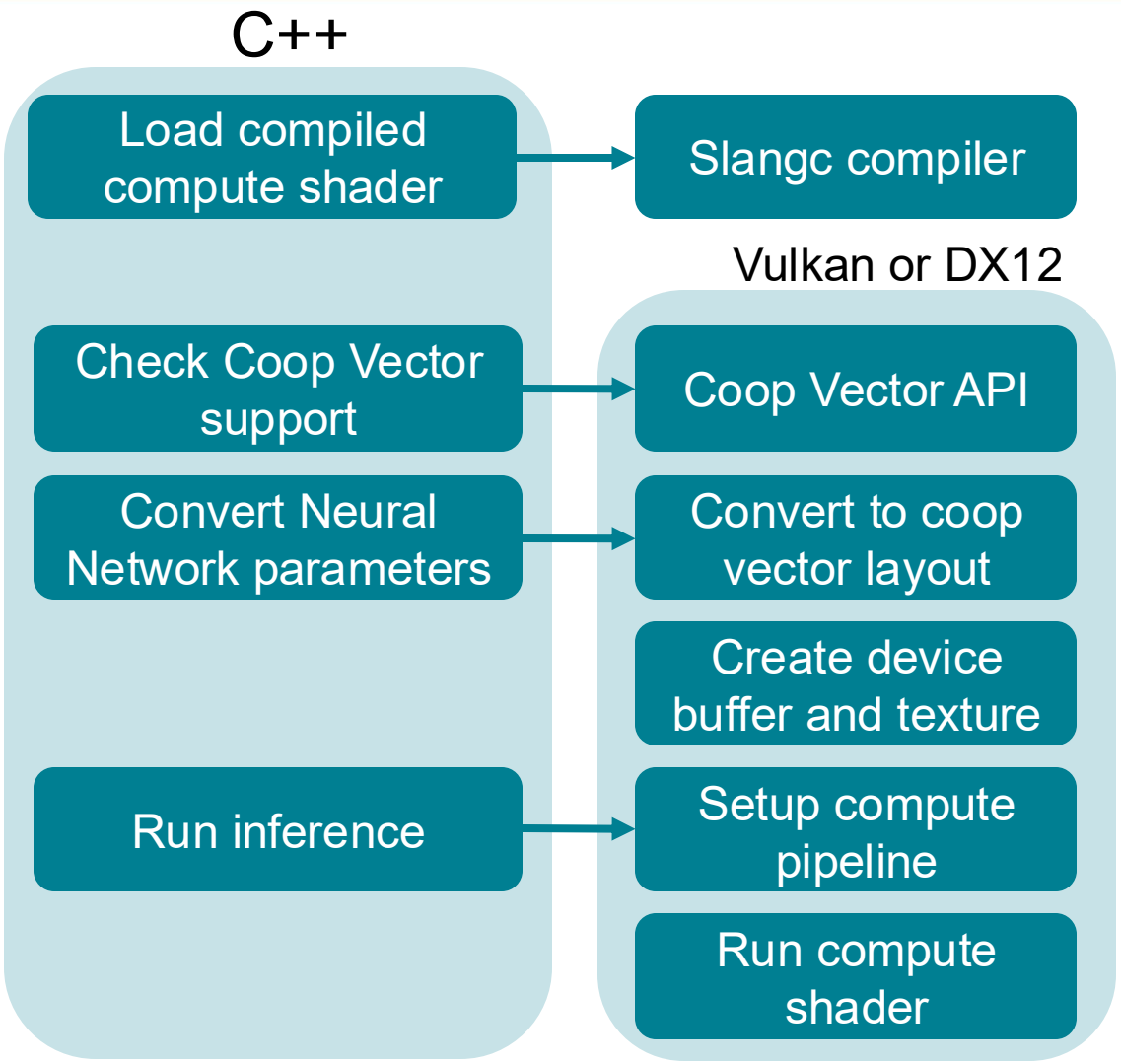
```python
# Load values of biases and weights
weights_np = np.array(data['weights'], dtype=np.float16).reshape((outputs, inputs))
biases_np = np.array(data['biases'], dtype=np.float16)

# Convert weights into coopvec layout
desc = app.device.coopvec_create_matrix_desc(self.outputs, self.inputs,
                                             self.layout,
                                             spy.DataType.float16, 0)
weight_count = desc.size // 2 # sizeof(half)
params_np = np.zeros((weight_count, ), dtype=np.float16)
app.device.coopvec_convert_matrix_host(weights_np, params_np,
                                       dst_layout=self.layout)
```

```python
self.biases = app.device.create_buffer(struct_size=2, element_count=self.outputs,
                                       data=biases_np)
self.weights = app.device.create_buffer(struct_size=2, element_count=weight_count,
                                        data=params_np)
```

```python
lr_output = spy.Tensor.empty_like(image)
module.inference(pixel = spy.call_id(),
                 resolution = res,
                 network = network,
                 _result = lr_output)
```

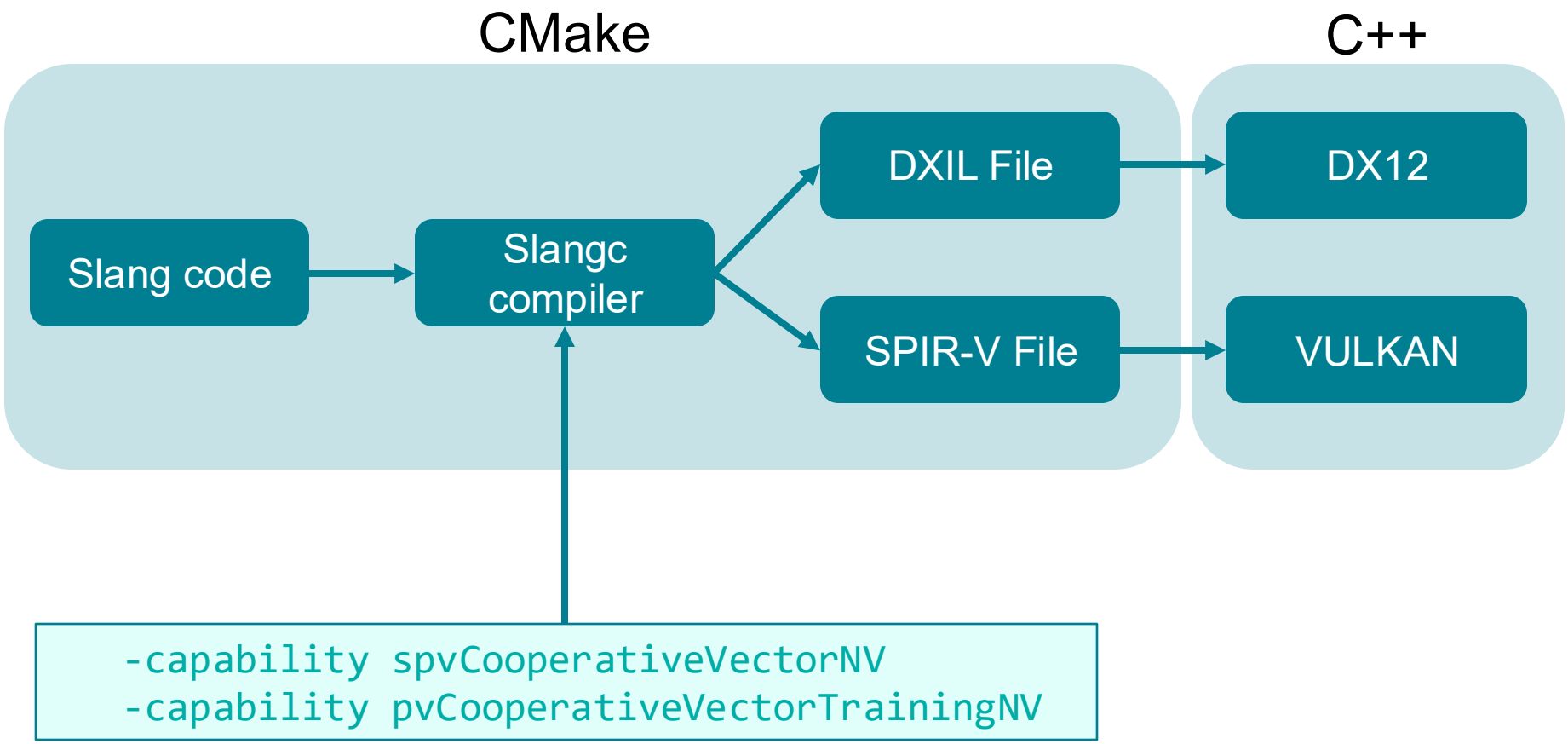
# C++ IMPLEMENTATION

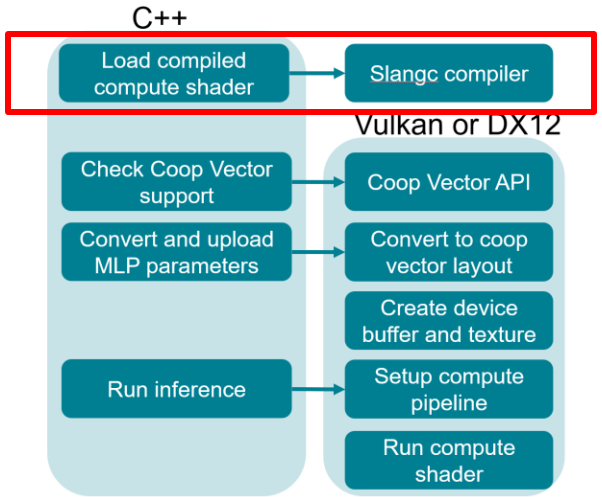




# STEP 1 – SLANG AS SHADER MODULE

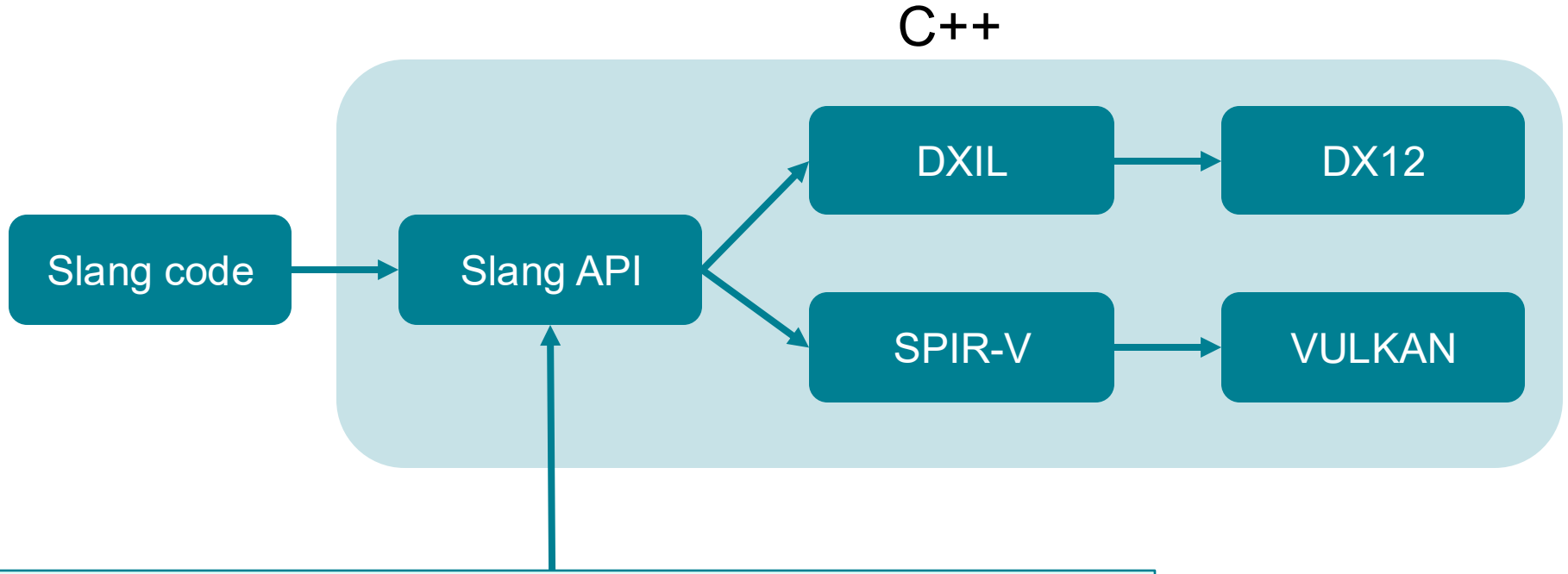




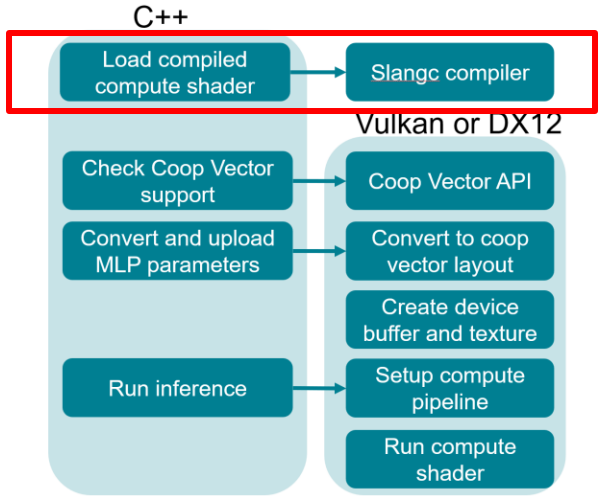



# STEP 1 – SLANG AS SHADER MODULE






- Cooperative Vector parameters
  - -capability spvCooperativeVectorNV
  - -capability pvCooperativeVectorTrainingNV
- Compile time parameters
- Template parameters
- Reflection



# STEP 2 - FEATURE DETECTION FOR COOPERATIVE VECTORS (VULKAN)



Required Vulkan device extensions:

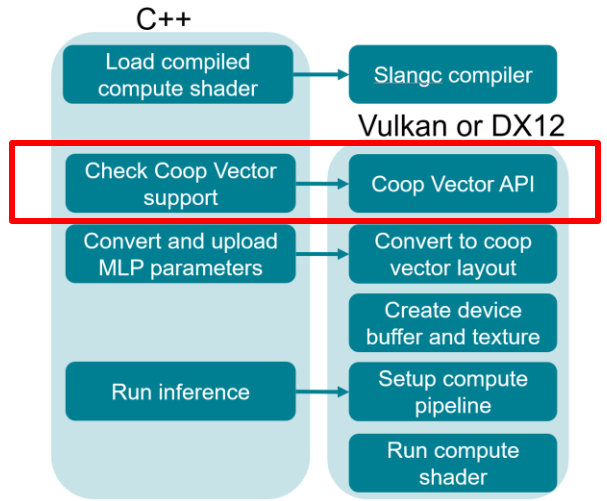
`VK_NV_cooperative_vector`

Required Vulkan physical device features:

`VK_STRUCTURE_TYPE_PHYSICAL_DEVICE_COOPERATIVE_VECTOR_FEATURES_NV`

`cooperativeVector = true`

`cooperativeVectorTraining = true`



# STEP 2 - FEATURE DETECTION FOR COOPERATIVE VECTORS (DX12)

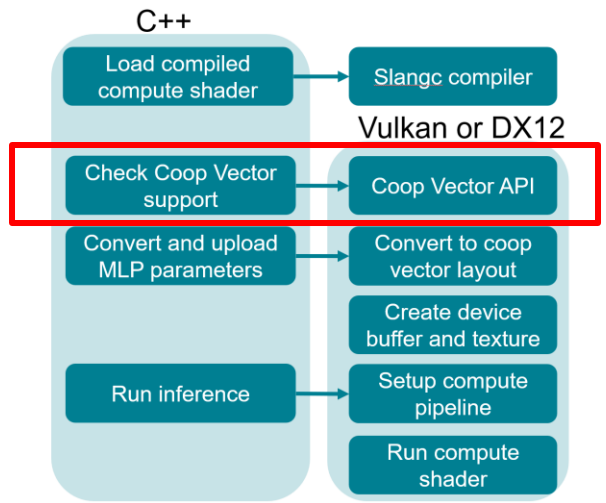


Call `D3D12EnableExperimentalFeatures` to enable following features:

- `D3D12ExperimentalShaderModels`
- `D3D12CooperativeVectorExperiment`

Call D3D12 device method `CheckFeatureSupport` for `D3D12_FEATURE_D3D12_OPTIONS_EXPERIMENTAL` and check:

- `D3D12_COOPERATIVE_VECTOR_TIER_1_0` for inference support
- `D3D12_COOPERATIVE_VECTOR_TIER_1_1` for training support

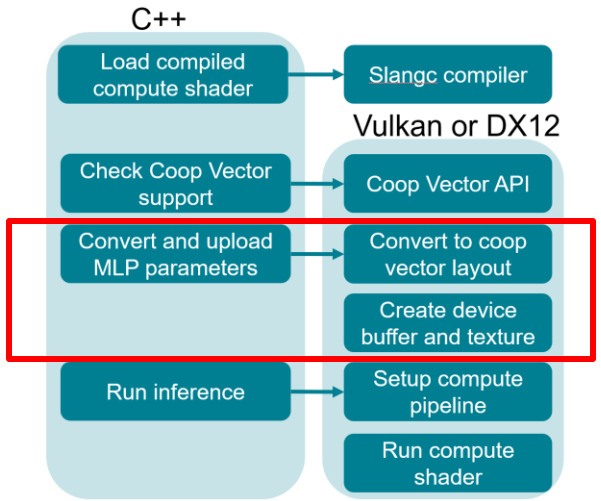


# STEP 3 – MATRIX OPTIMAL LAYOUTS



## FEED-FORWARD LAYER USING COOPERATIVE VECTORS

```
struct NetworkParameters<int Inputs, int Outputs>
{
    ByteAddressBuffer weights, biases;

    CoopVec<half, Outputs> forward(CoopVec<half, Inputs> x)
    {
        return coopVecMatMulAdd<half, Outputs>(
            x, CoopVecComponentType.Float16,
            weights, 0, CoopVecComponentType.Float16,
            biases, 0, CoopVecComponentType.Float16,
            CoopVecMatrixLayout.InferencingOptimal,
            false,
            0
        );
    }
}
```

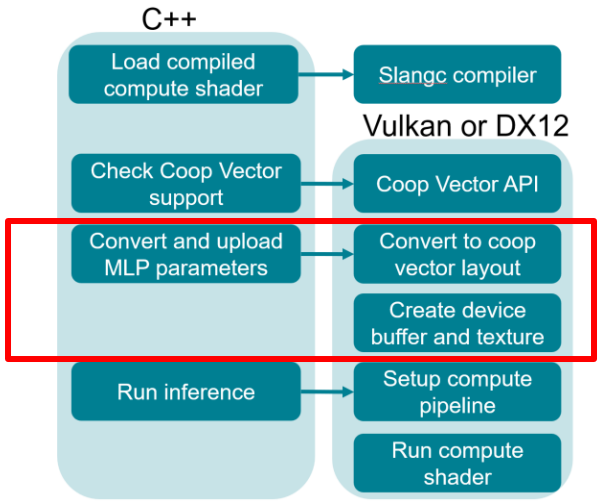



- Tensor cores use custom matrix layouts
  - Components of matrices are shuffled between threads
  - Specific layout depends on the GPU and data types

# STEP 3 – MATRIX LAYOUT CONVERSION



1. Create parameters buffer in optimal layout
   - Query shuffled matrix size from VK/DX12
   - Upload row- or column-major matrix to GPU
   - Convert layout (perform the shuffle) on the GPU
     - Inference Optimal
     - Training Optimal




**Vulkan**

- `vkConvertCooperativeVectorMatrixNV`
  - CPU conversion / size query
- `vkCmdConvertCooperativeVectorMatrixNV`
  - GPU conversion

**DX12**

- `GetLinearAlgebraMatrixConversionDestinationInfo`
  - Size query
- `ConvertLinearAlgebraMatrix`
  - GPU conversion

2. Create constant buffer
3. Create output texture

# STEP 4 – CONVERT INFERENCE FUNCTION TO SHADER



## SLANG

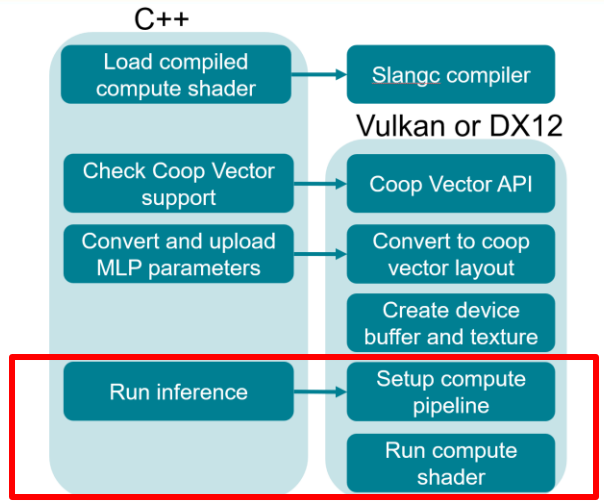
```
struct NetworkParameters<int Inputs, int Outputs> {
    ByteAddressBuffer<half> weights, biases;

    CoopVec<half, Outputs> forward(CoopVec<half, Inputs> x);
}
```

```
struct Network {
    NetworkParameters<16, 32> layer0, layer1, layer2;

    float3 eval(no_diff float2 uv);
}
```

```
float3 inference(int2 pixel, int2 resolution, Network network)
{
    float2 uv = (float2(pixel) + 0.5f) / float2(resolution);
    return network.eval(uv);
}
```

## C++

```
lr_output = spy.Tensor.empty_like(image)

module.inference(pixel = spy.call_id(), resolution = res, network = network, _result = lr_output)
```




## SLANGPY

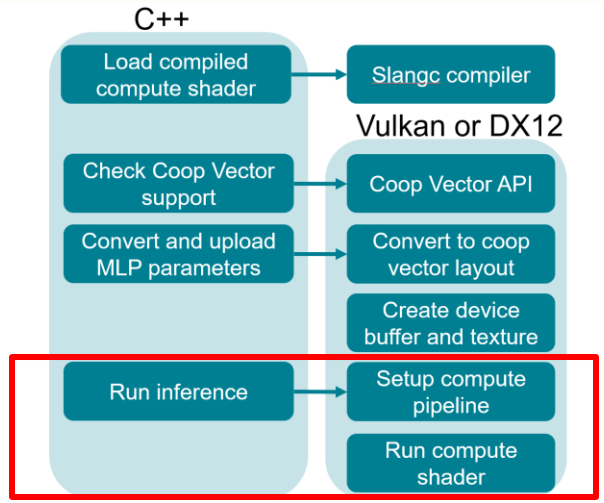
Compute Shader

# STEP 4 – CONVERT INFERENCE FUNCTION TO SHADER



## SLANG

```
struct NetworkParameters<int Inputs, int Outputs> {
    ByteAddressBuffer <half> weights, biases;

    CoopVec<half, Outputs> forward(CoopVec<half, Inputs> x);
}

struct Network {
    NetworkParameters<16, 32> layer0;
    NetworkParameters<32, 32> layer1;
    NetworkParameters<32, 3>  layer2;

    float3 eval(no_diff float2 uv);
}

float3 inference(int2 pixel, int2 resolution, Network network)
{
    float2 uv = (float2(pixel) + 0.5f) / float2(resolution);
    return network.eval(uv);
}
```




```
Network network;

ConstantBuffer<NeuralConstants> gConst;
RWTexture2D<float4> outputTexture;

[shader("compute")]
[numthreads(8, 8, 1)]
void main_cs(uint3 pixel : SV_DispatchThreadID)
{
    outputTexture[pixel.xy] = inference(pixel.xy,
                                        gConst.resolution,
                                        network);
}
```

# STEP 4 – CONVERT INFERENCE FUNCTION TO SHADER

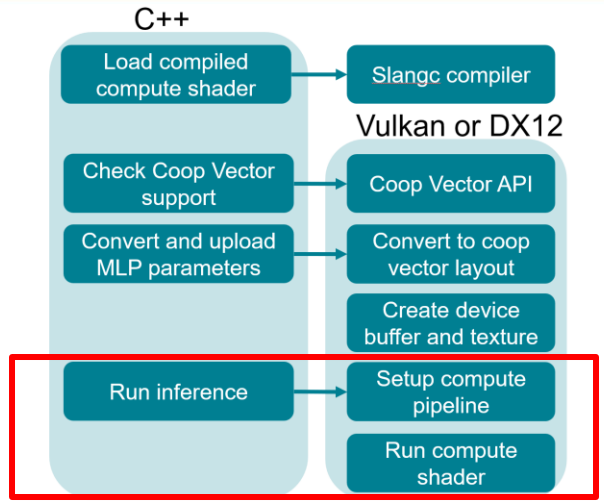


## SLANG

```
struct NetworkParameters<int Inputs, int Outputs> {
    ByteAddressBuffer <half> weights, biases;
    CoopVec<half, Outputs> forward(CoopVec<half, Inputs> x);
}

struct Network {
    NetworkParameters<16, 32> layer0;
    NetworkParameters<32, 32> layer1;
    NetworkParameters<32, 3>  layer2;

    float3 eval(no_diff float2 uv);
}

Network network;

ConstantBuffer<NeuralConstants> gConst;
RWTexture2D<float4> outputTexture;

[shader("compute")]
[numthreads(8, 8, 1)]
void main_cs(uint3 pixel : SV_DispatchThreadID){
 // Compute shader
}
```

## C++

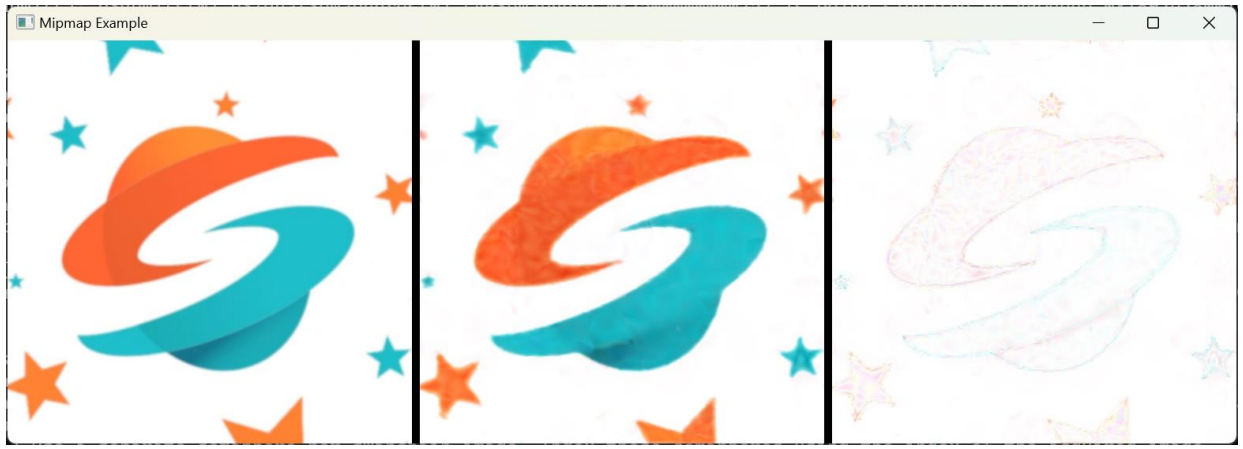
```
// Initialize parameters buffers

// Setup up constant buffer

// Bind output texture

// Dispatch compute shader
nvrhi::ComputeState state;
m_CommandList->setComputeState(state);
m_CommandList->dispatch(textureWidth, textureHeight, 1);
```



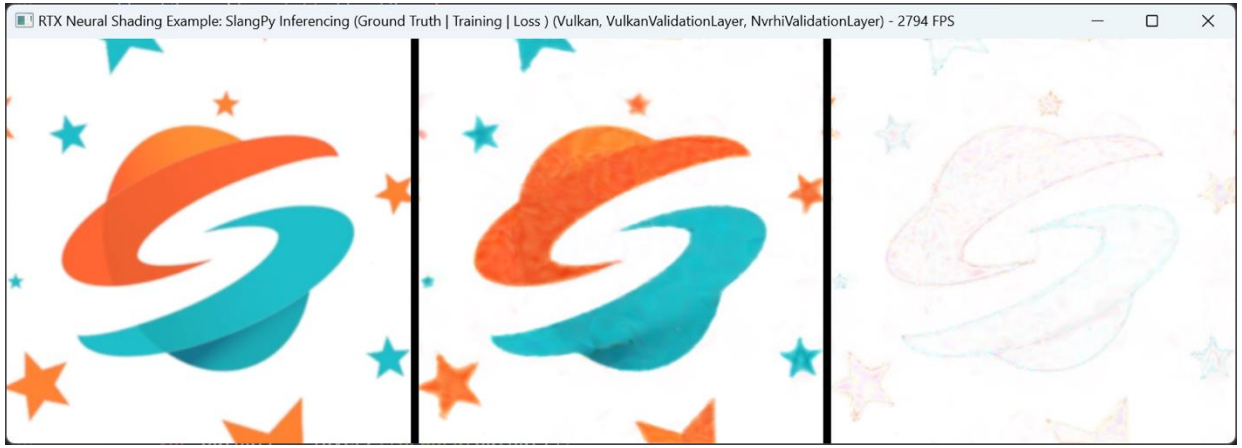
# YOUR NEURAL SHADER IS NOW SHIPPED!



Python




C++

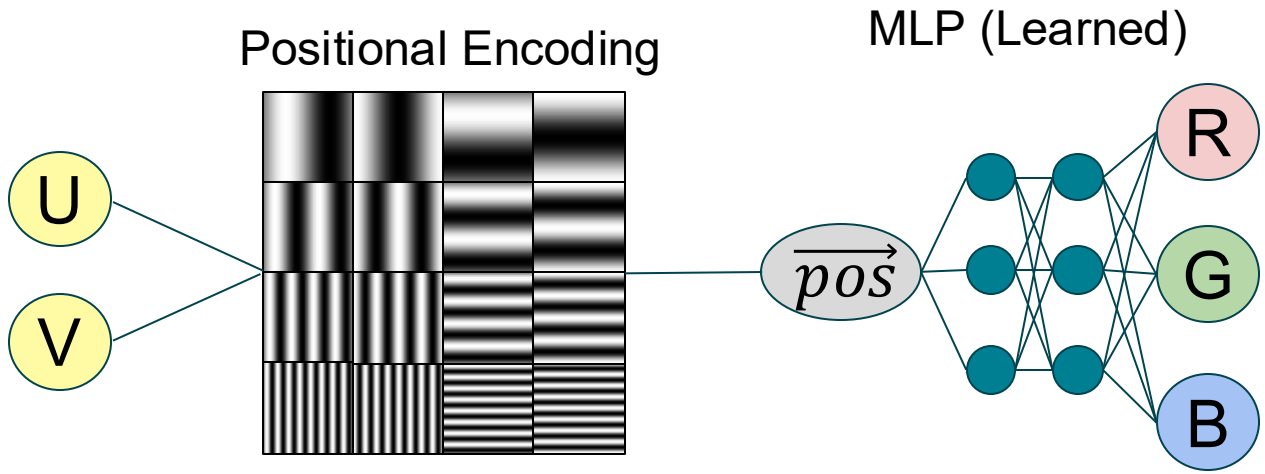


SIGGRAPH 2025
Vancouver+ 10-14 August

# NEURAL TEXTURE COMPRESSION

# FLASHBACK: PREVIOUS BEST RESULT

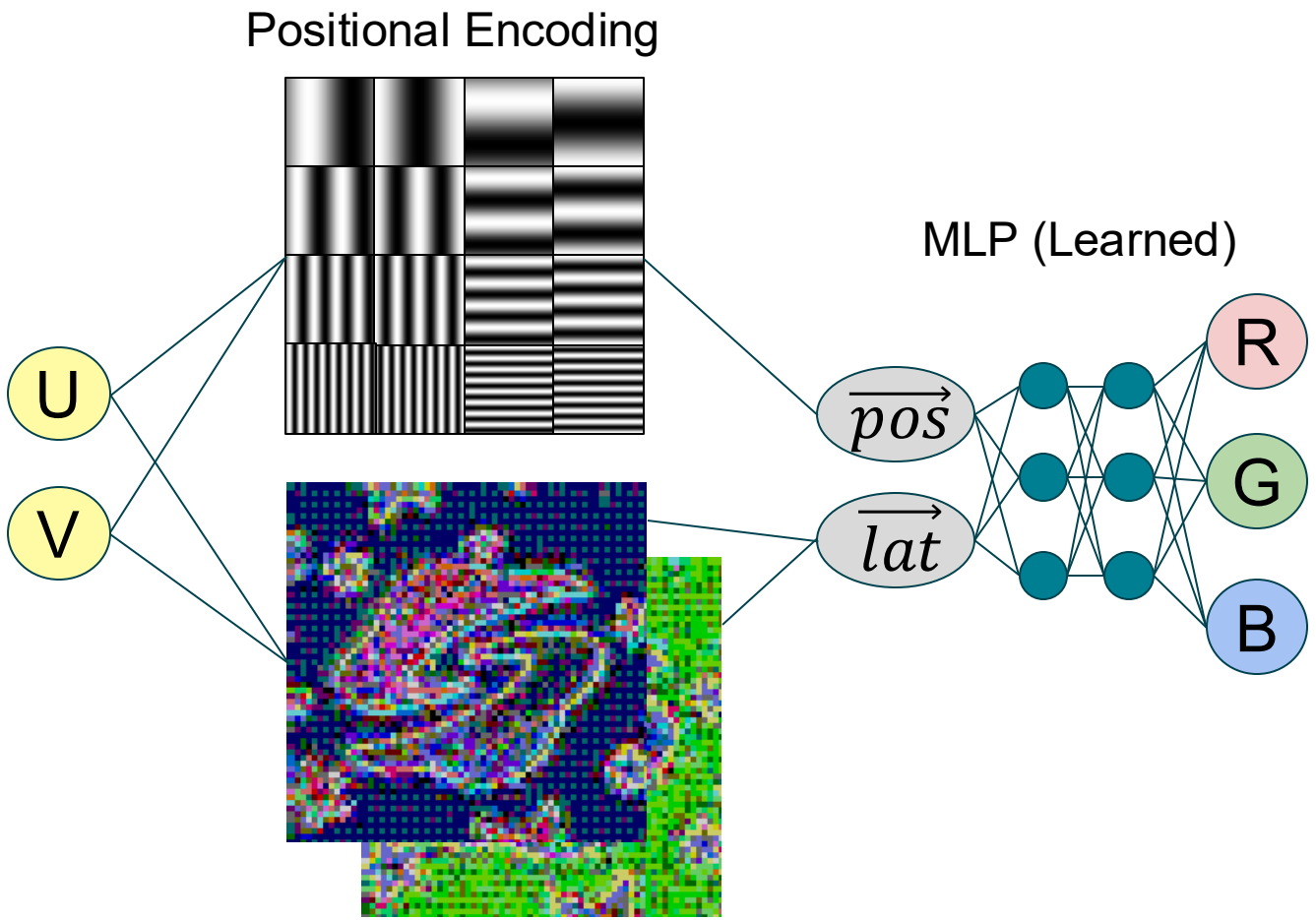





Output Image (meh)

# ADDING LATENT TEXTURES





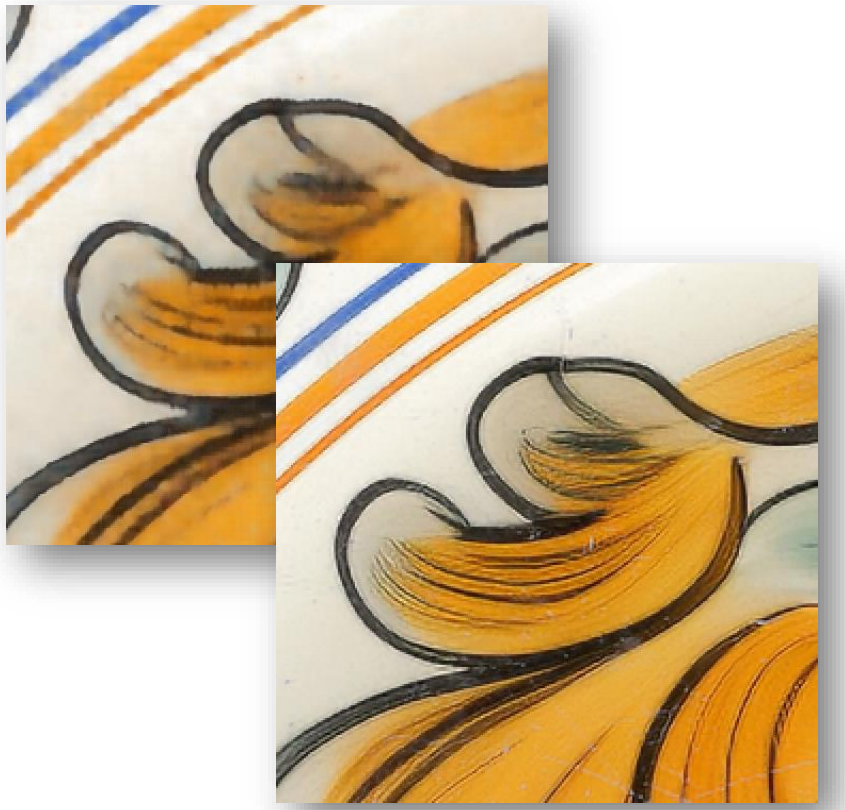

Output Image (pretty good)

# KEY FACTS ABOUT NTC



- Wide range of latent texture parameters
  - Independent resolution, channel count, bits per feature controls
  - Overall practical range of 0.5 — 20 bits per pixel storing up to 16 channels
  - MLP weights are tiny compared to the latents (~12 KB vs. many MB)
- Each texel can be decoded independently
  - Suitable for use as a replacement for regular material textures
- Neural compression without hallucinations
  - Compression is training of the network and latent textures
  - Network is small and overfitted to reproduce only one material

- Original paper describing NTC and STF:
  - K. Vaidyanathan et al. “Random-Access Neural Compression of Material Textures”

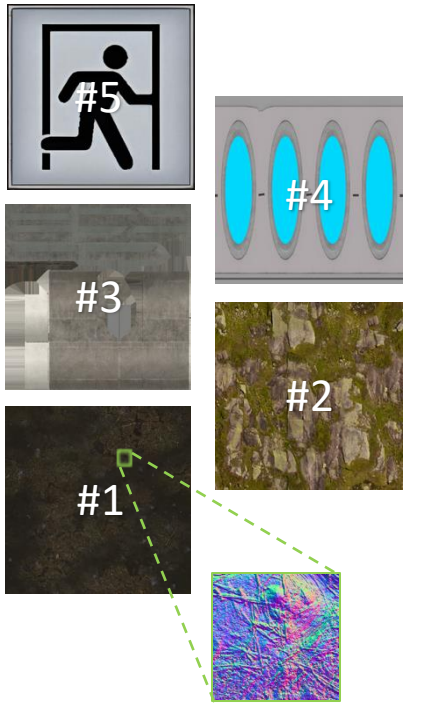


*Crops from an NTC compressed texture at 0.5 and 20.0 bpp*

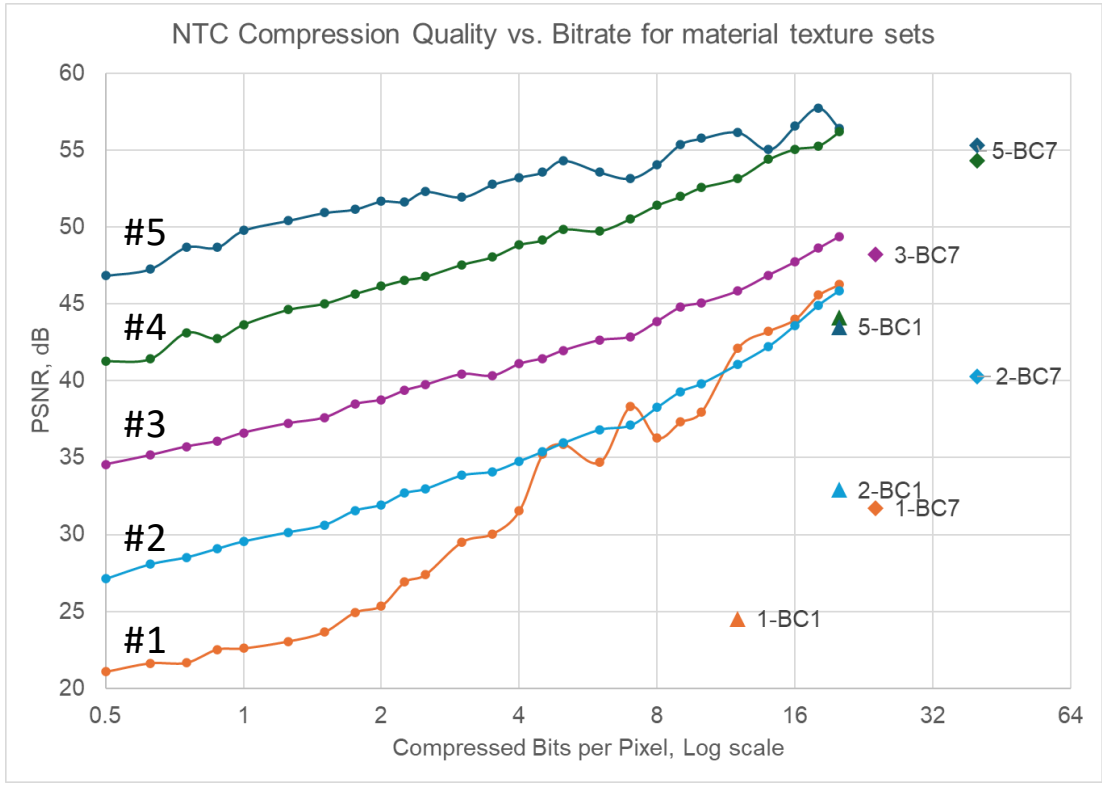
# NEURAL TEXTURE COMPRESSION QUALITY



- Consistently better than BCn
  - More benefit for complex materials
  - Up to 8x better at similar visual quality
  - Introduces blur, not blockiness
- Mostly monotonic vs. BPP
  - More information means higher quality
  - Training finds effective ways to use the bits
- Benefits from correlated channels
  - All textures for a PBR material are better compressed together than separately



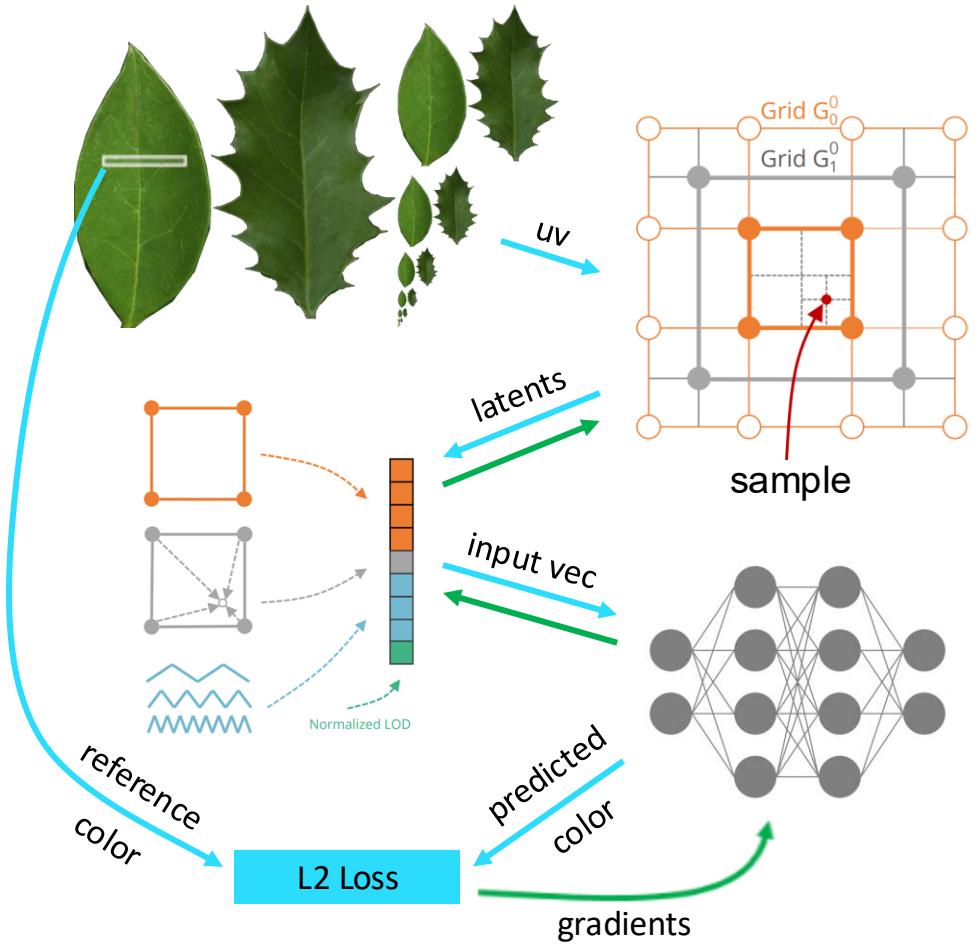

*High detail normal map is hard to compress*



# NTC TRAINING PROCESS



- Pick a set of pixels to process on this step
  - 64k pixels per step is usually good
- For each pixel in parallel:
  - Calculate positional encoding based on UV
  - Sample the latent textures at UV
  - Prepare MLP input vector
  - Evaluate MLP
  - Load reference texture channels
  - Compute gradient ~= (predicted – reference)
  - Backpropagate gradients
  - Accumulate gradients for MLP and features
- Run an optimizer step
  - Add the quantization noise
- Repeat 100-200k times

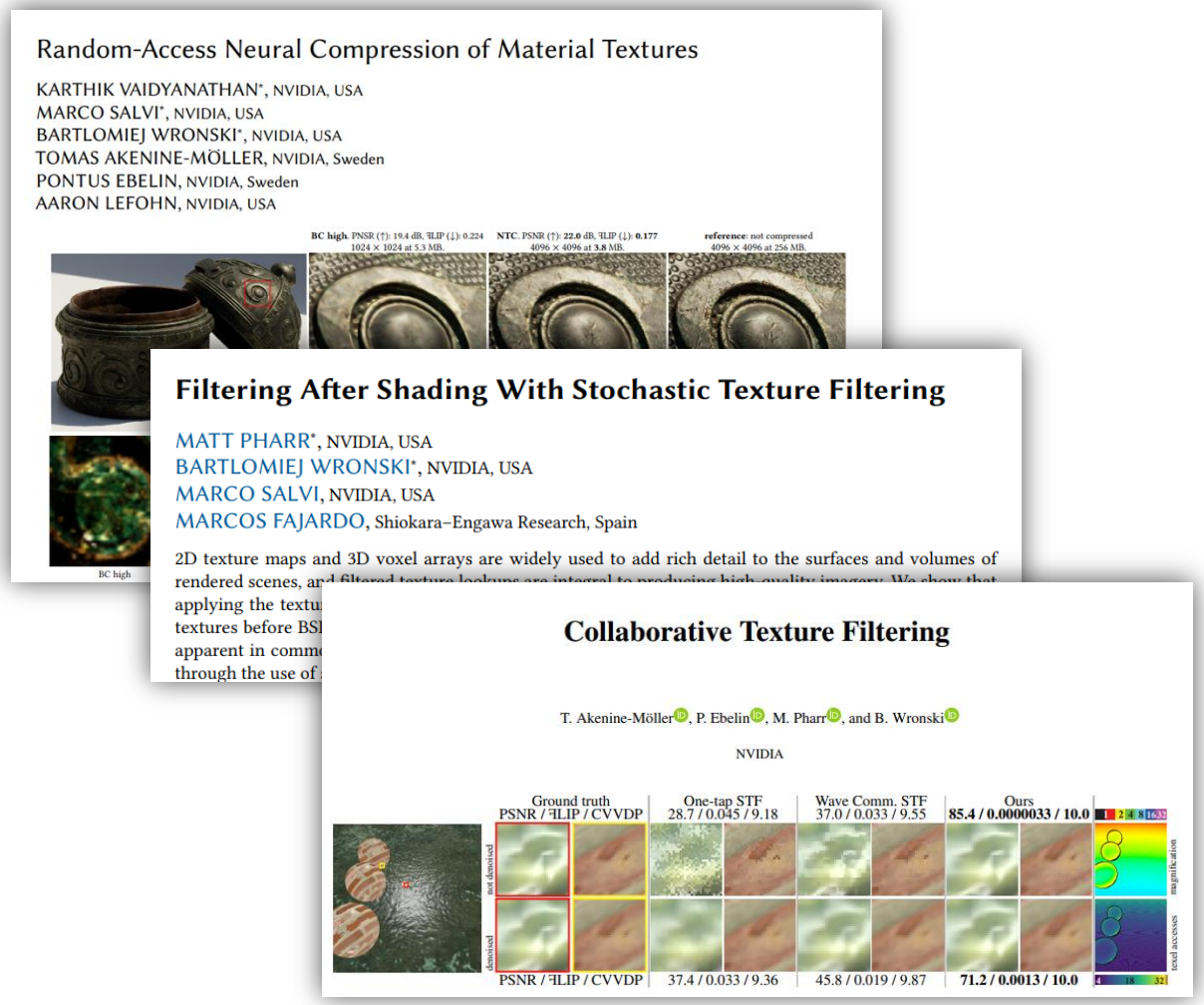


# NTC MODES OF OPERATION 1/3



## INFERENCE ON SAMPLE

- Replace material texture sampling with NTC decode
- Apply Stochastic Texture Filtering (STF) instead of hardware filtering
  - NTC gives one pixel per decode, direct trilinear or aniso filtering would be too expensive
- Benefits:
  - Minimal VRAM footprint
- Drawbacks:
  - Makes shaders larger and slower (depends…)
  - STF is a requirement

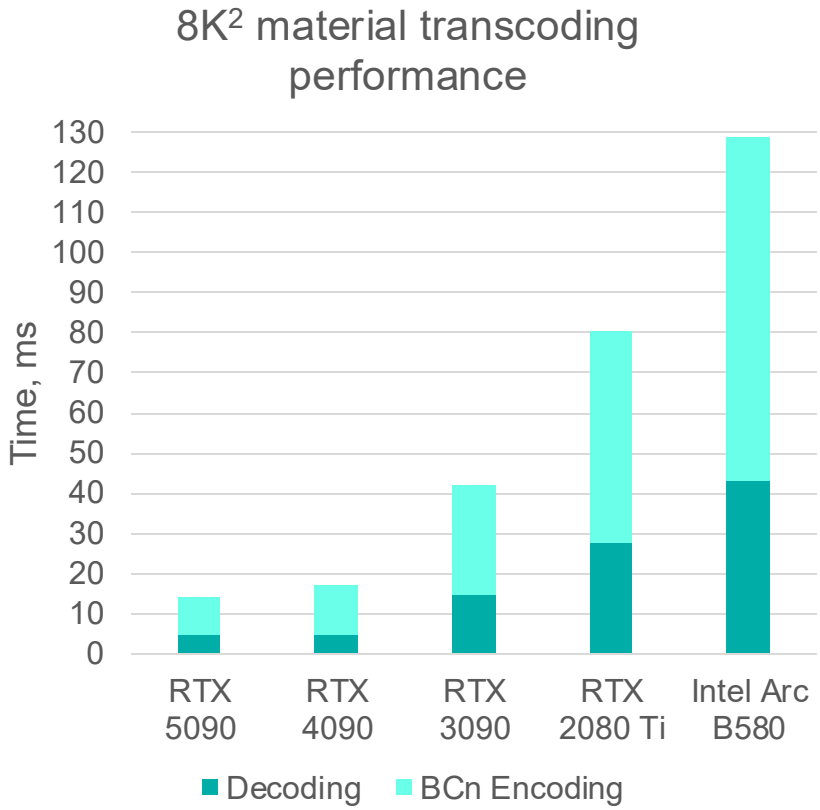


# NTC MODES OF OPERATION 2/3



## INFERENCE ON LOAD

- Transcode NTC materials into BCn at load time
- Sample BCn textures as usual
- Benefits:
  - Easy integration
  - Rendering performance is unaffected
- Drawbacks:
  - No VRAM savings, only disk space / network traffic




*Using FP8 inference with CoopVec for NTC*

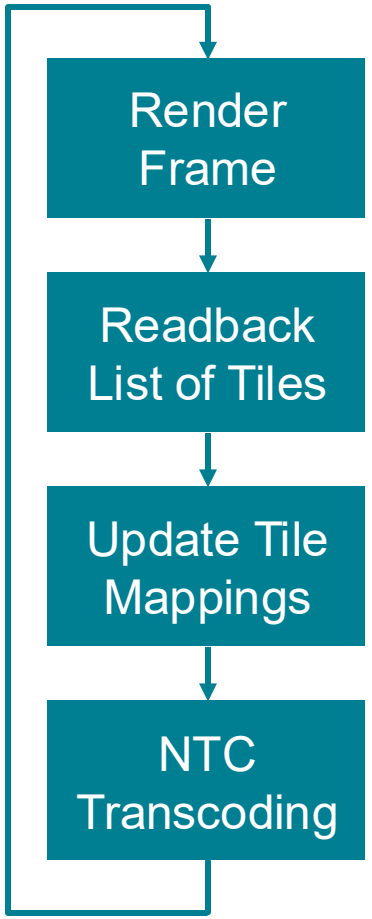
*Encoding 2x BC7 and 1x BC5*

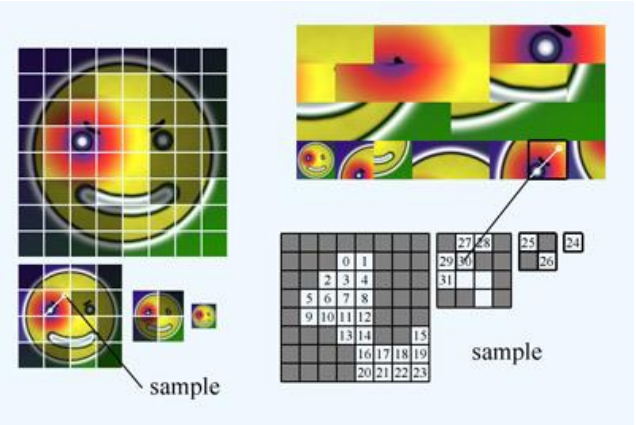
# NTC MODES OF OPERATION 3/3



## INFERENCE ON FEEDBACK

- Extension for a virtual texturing system
- Track which texture MIPs and tiles are needed
- Transcode the necessary tiles to BCn at runtime
- Benefits:
  - Low performance impact on render passes
- Drawbacks:
  - VRAM usage includes both NTC and partial BCn data
  - Potentially uneven frame times




*Virtual texture*

*Image: Sean Barrett, GDC 2008*

https://silverspaceship.com/src/svt/

# NTC PERFORMANCE COMPARISON




*Intel Sponza scene in the NTC SDK Renderer*

| Mode | Render Time | Texture Memory |
|---|---|---|
| On Load | 0.39 ms | 2041 MB |
| On Sample | 0.82 ms | 243 MB |
| On Feedback | 0.65 ms | 555 MB |

*Measured on an RTX PRO 6000 Blackwell WE*

# BENEFITS OF NEURAL TEXTURE COMPRESSION



## PRACTICAL

- Saves disk space
- Saves download traffic
- Saves VRAM (maybe)
- Can be used on any platform
  - High-end PC – on-sample
  - Low-end and consoles – on-load or on-feedback
- Can be used now
  - SDK available: github.com / NVIDIA-RTX / RTXNTC

## CONCEPTUAL

- Allows using higher detail materials
  - Fewer bits per pixel => more pixels with the same storage
- Can be extended with perceptual loss functions
  - Higher compression ratios with better visual detail
  - Ongoing research direction

# MORE NEURAL COMPRESSION TECHNIQUES



**Real-Time Neural Materials using Block-Compressed Features**

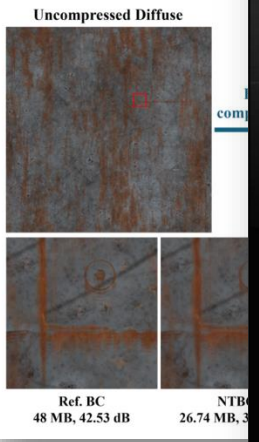
C. Weinreich†, L. De Oliveira†, A. Houdard† and G. Nader†

Ubisoft La Forge

**Neural Texture Block Compression**

S. Fujieda and T. Harada

Advanced Micro Devices, Inc.




SIGGRAPH 2024
DENVER+ 28 JUL — 1 AUG

**NEURAL LIGHT GRID**

or adventures in trying to get something useful out of ML in production

Michal Iwanicki
Peter-Pike Sloan
Ari Silvennoinen
Peter Shirley



GDC MARCH 17-21, 2025 SAN FRANCISCO, CA

**Global Illumination In "Once Human":**
**A Hybrid Approach for 16km Open World**

Maode Shi, Lele Pan

NetEase Games NeoX

#GDC2025

LIGHTSPEED STUDIOS | GDC

Decoding Light:
**Neural Compression of Global Illumination**

**Luyan Cao**, LIGHTSPEED STUDIOS

SIGGRAPH 2025
Vancouver+ 10-14 August

# REAL-TIME NEURAL MATERIALS

# INSPIRATION: REAL MATERIALS ARE COMPLEX

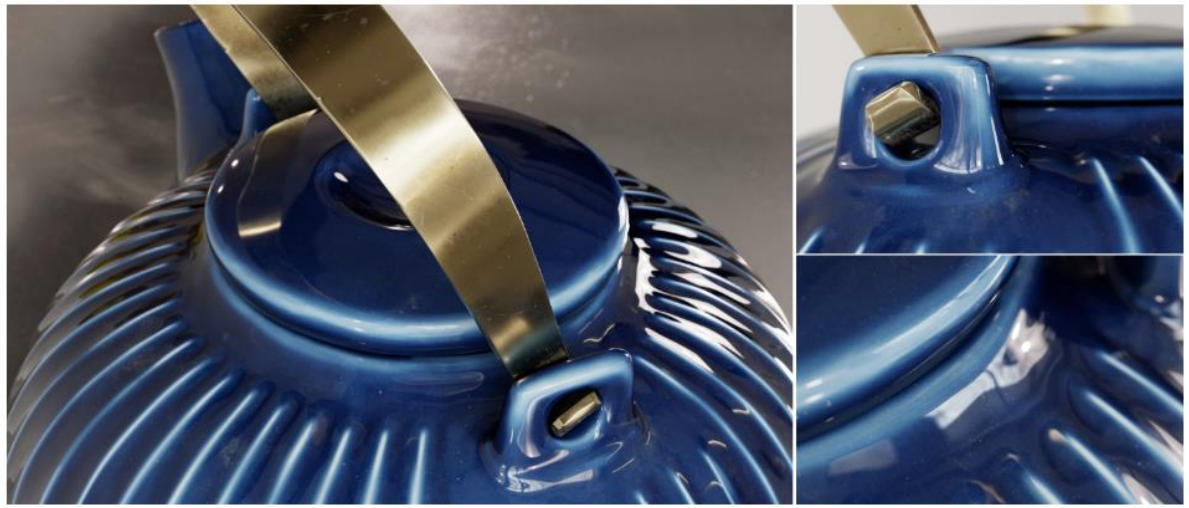


## Real-Time Neural Appearance Models

Tizian Zeltner[†] Fabrice Rousselle[†] Andrea Weidlich[†] Petrik Clarberg[†] Jan Novák[†]
Benedikt Bitterli[†] Alex Evans Tomáš Davidovič Simon Kallweit Aaron Lefohn

NVIDIA

[†]equal contribution, order determined by a rock-paper-scissors tournament.



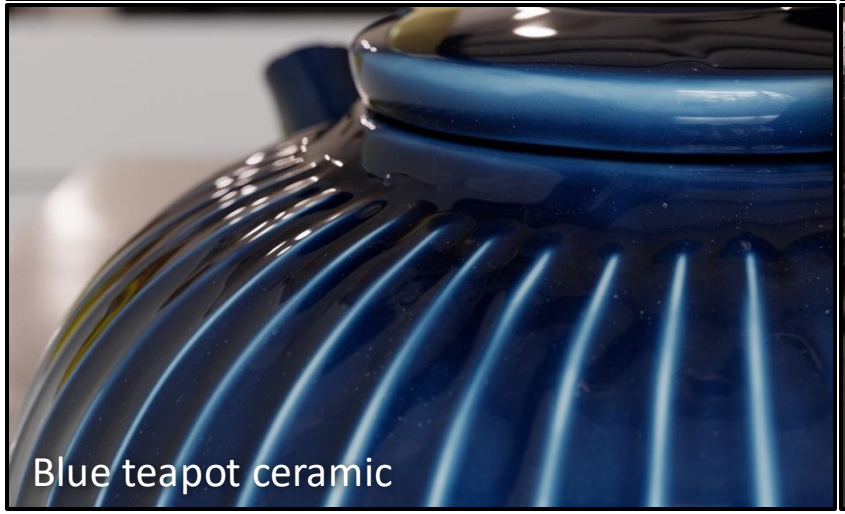
# INSPIRATION: REAL MATERIALS ARE COMPLEX



# WE CAN RENDER THESE, BUT NOT IN REAL-TIME

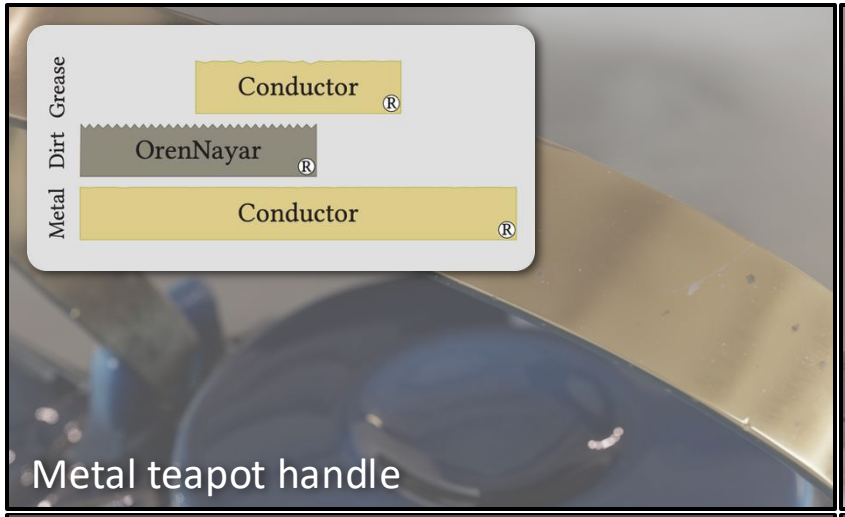


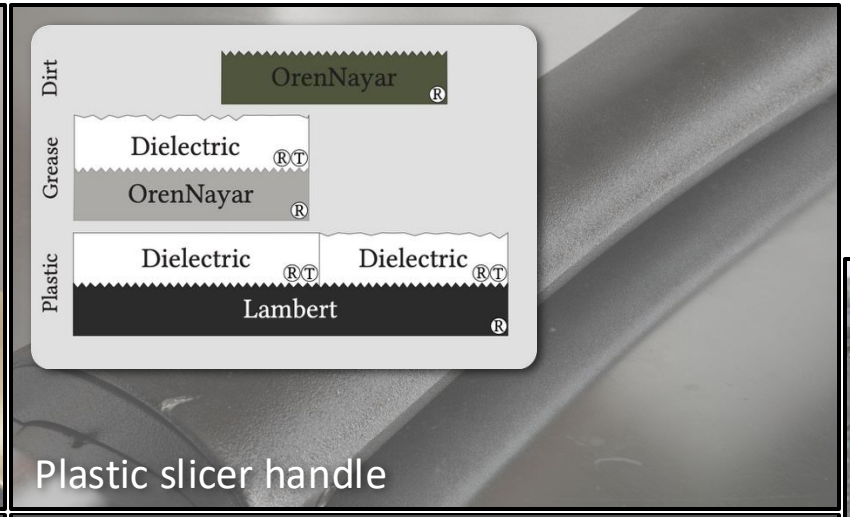


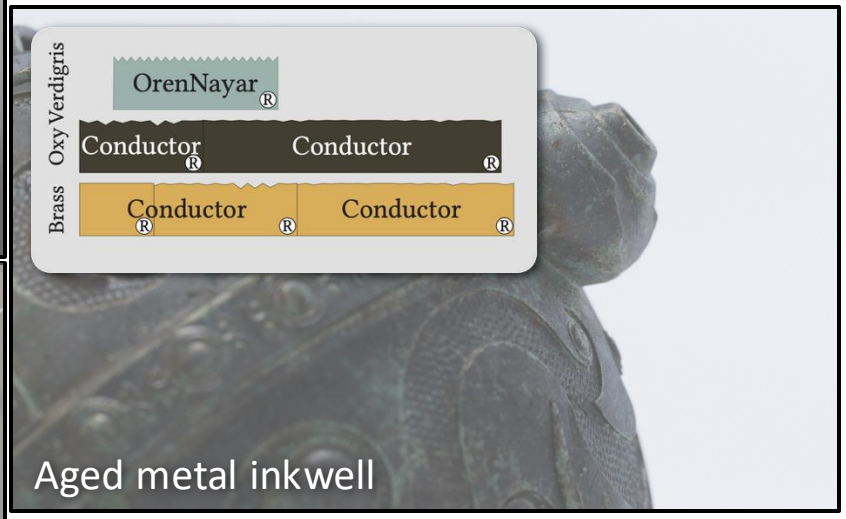




# THESE ARE COMPLEX MATERIAL GRAPHS…

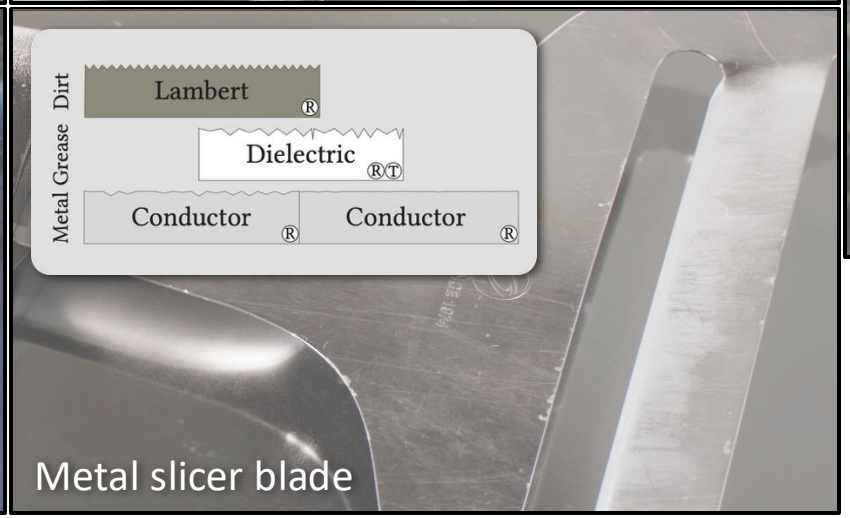




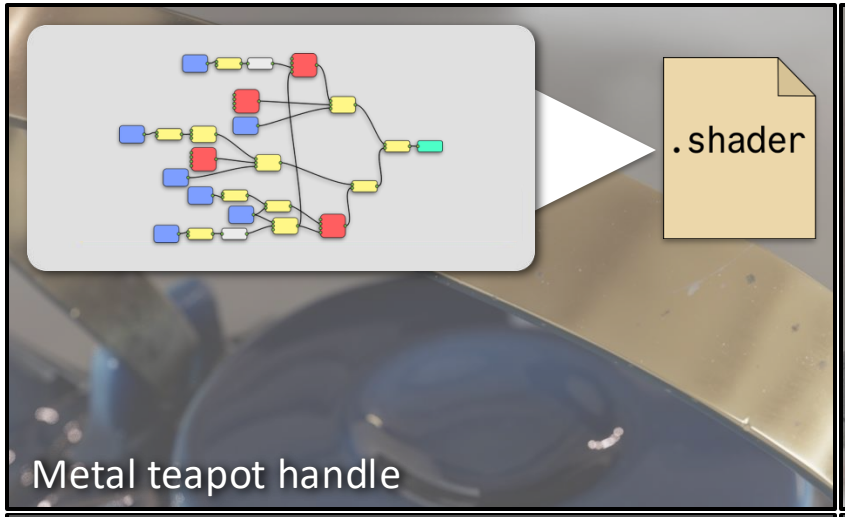



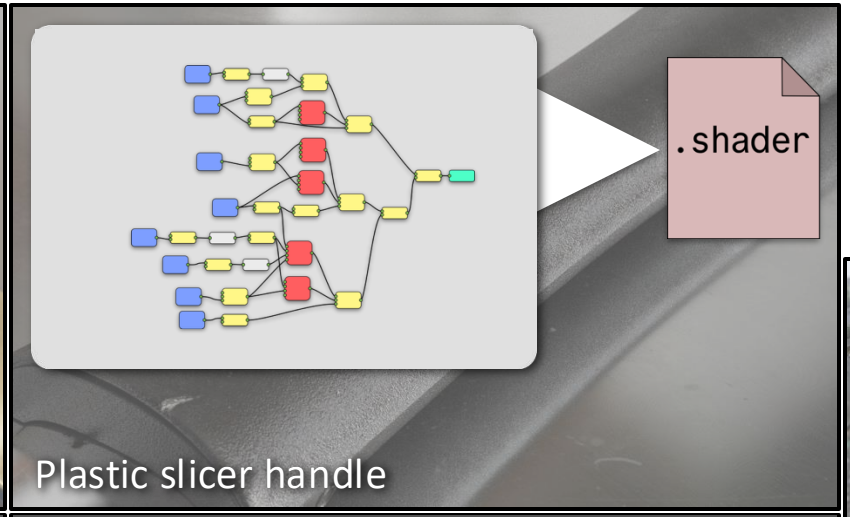


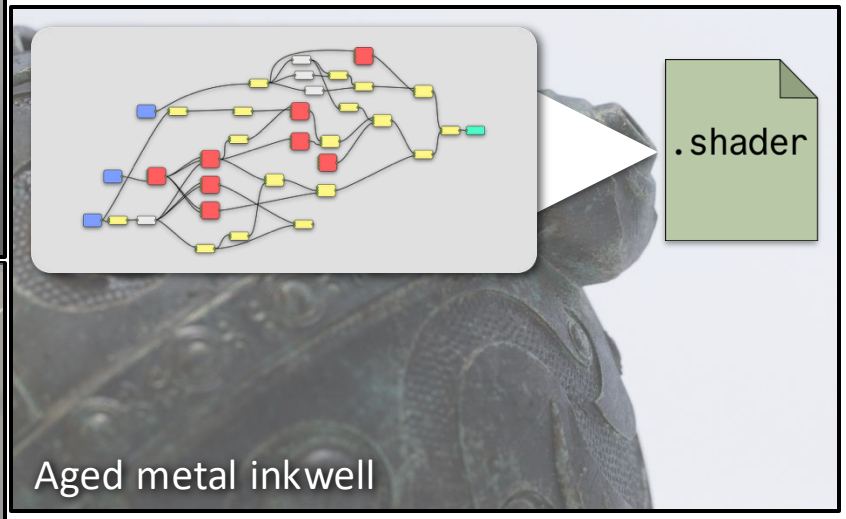


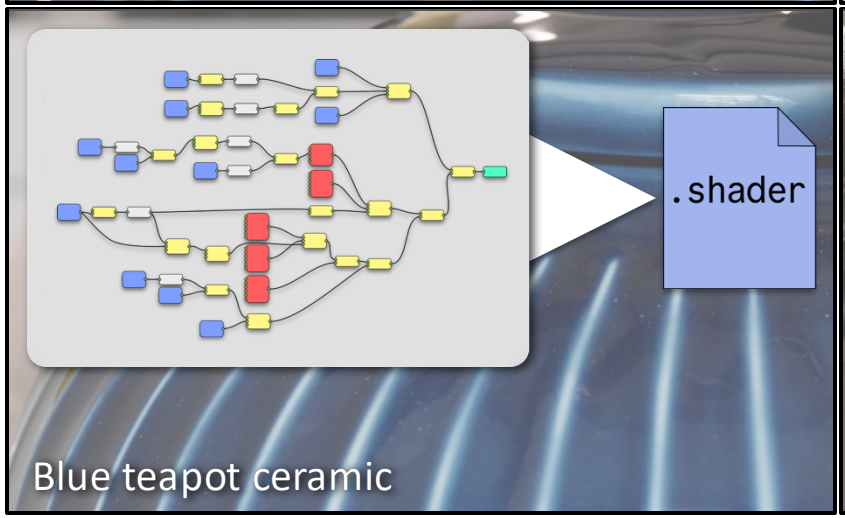
# …AND WE DON'T KNOW HOW TO SIMPLIFY THEM WELL

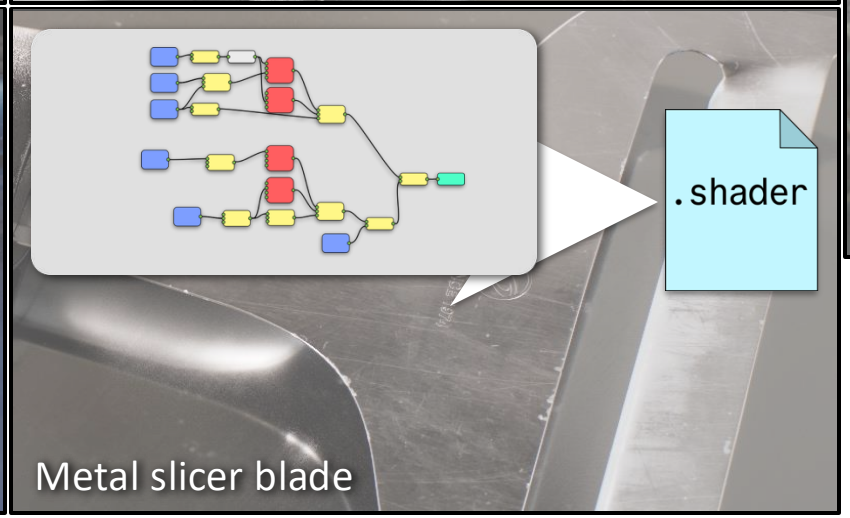




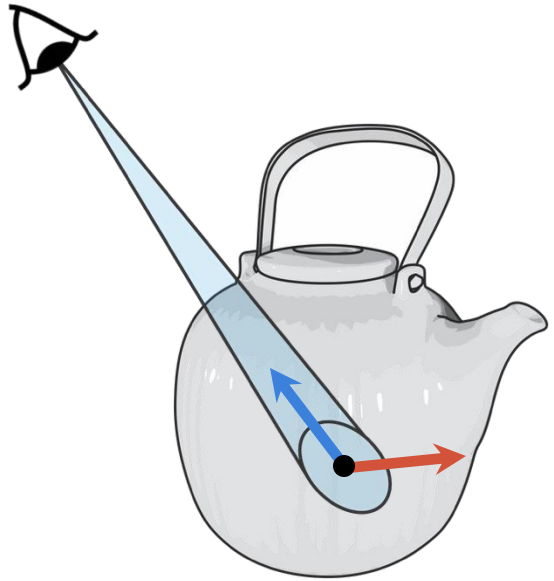



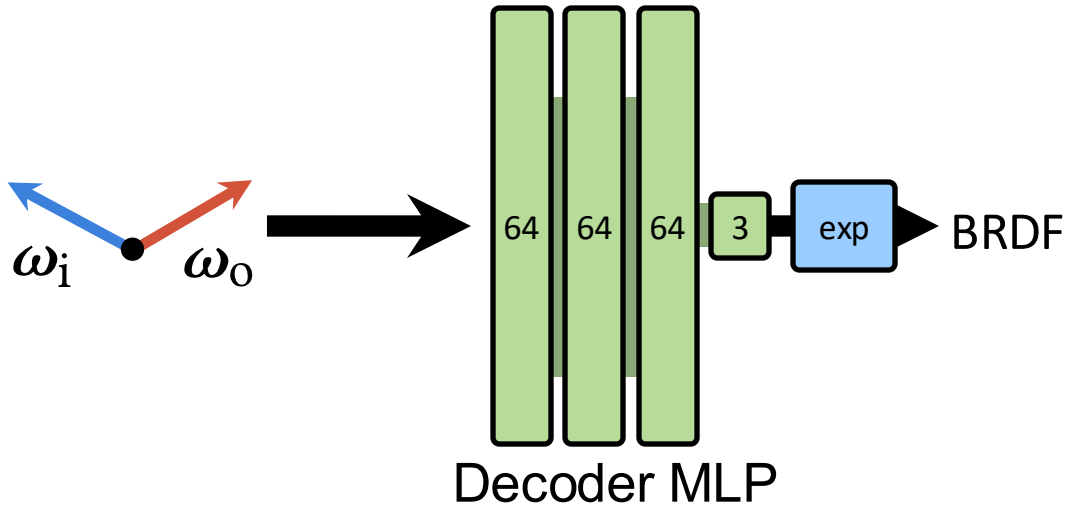



# WHAT IF WE USED A NEURAL NETWORK FOR THE TASK?

# SIMPLEST FIRST: PASS DIRECTIONS TO A NETWORK

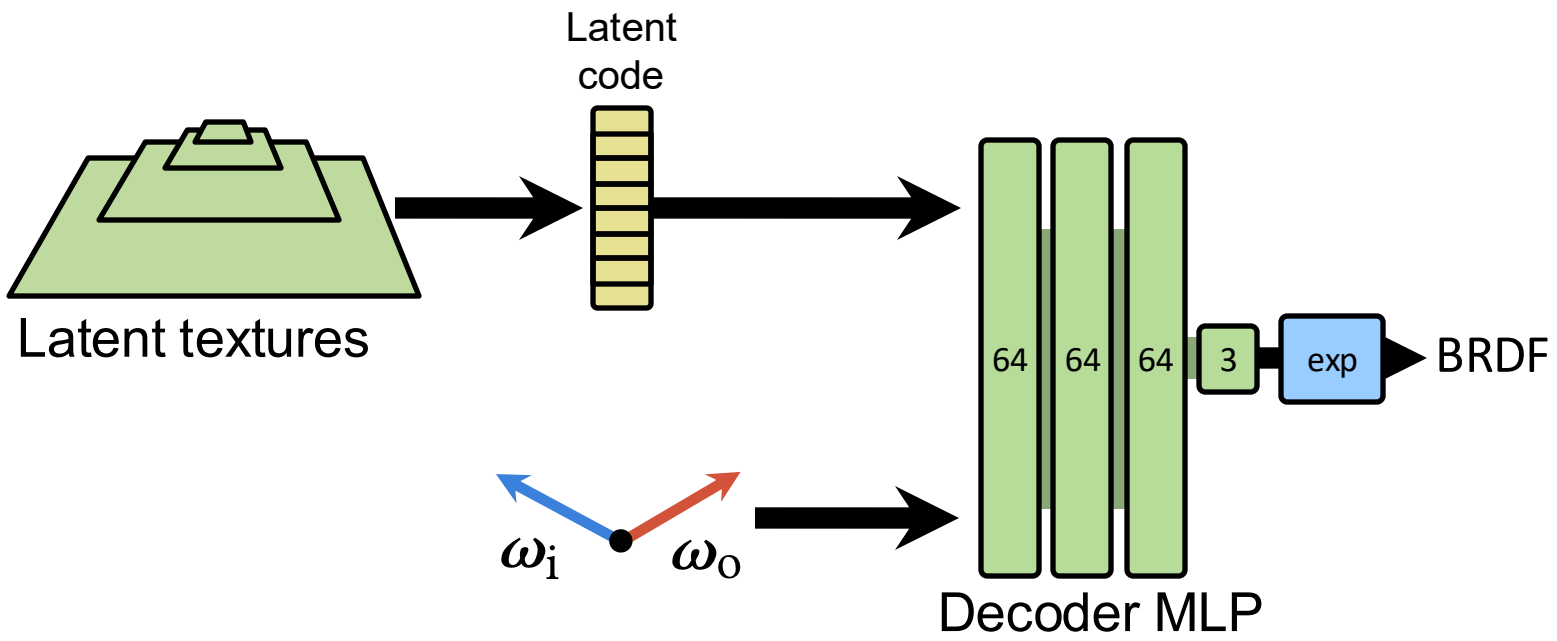






# …AND THEN TEXTURE IT



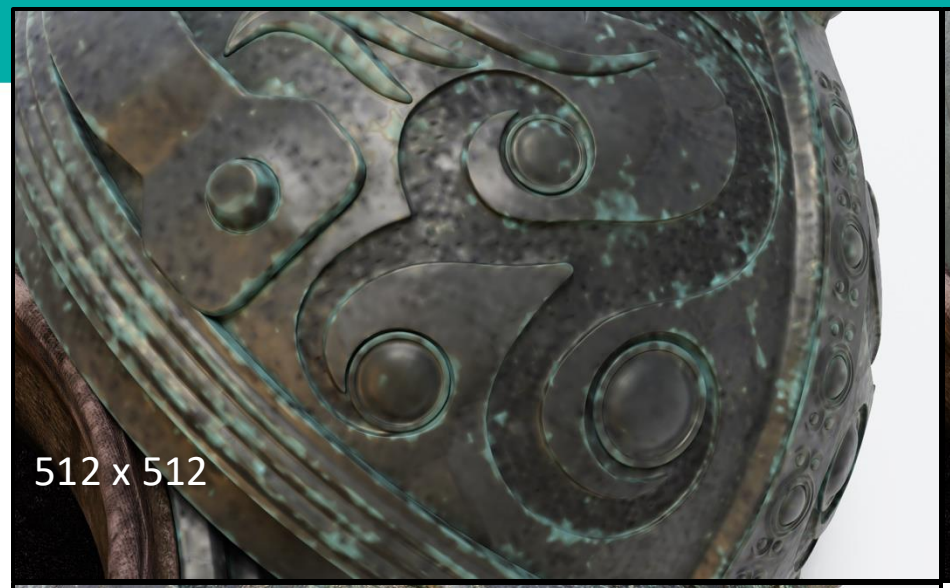


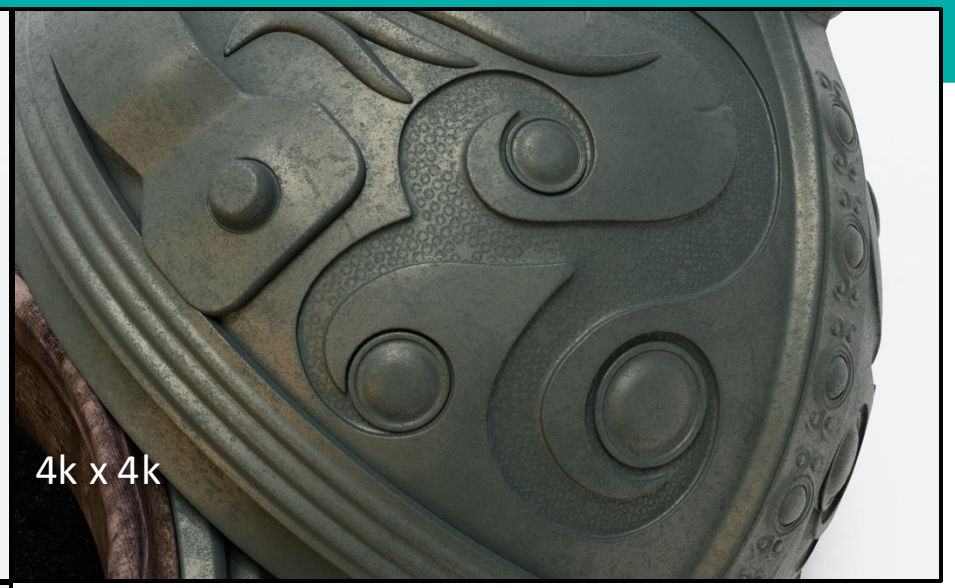
# THIS WORKS!

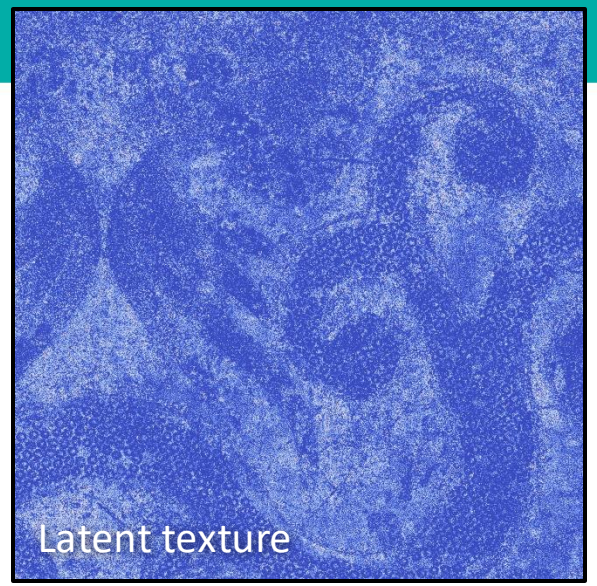


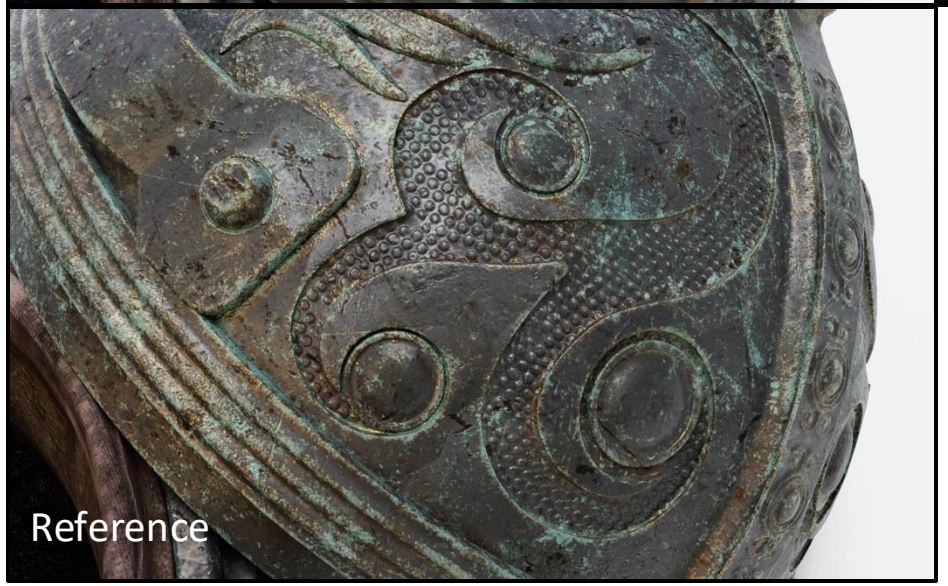
# …UP TO A POINT





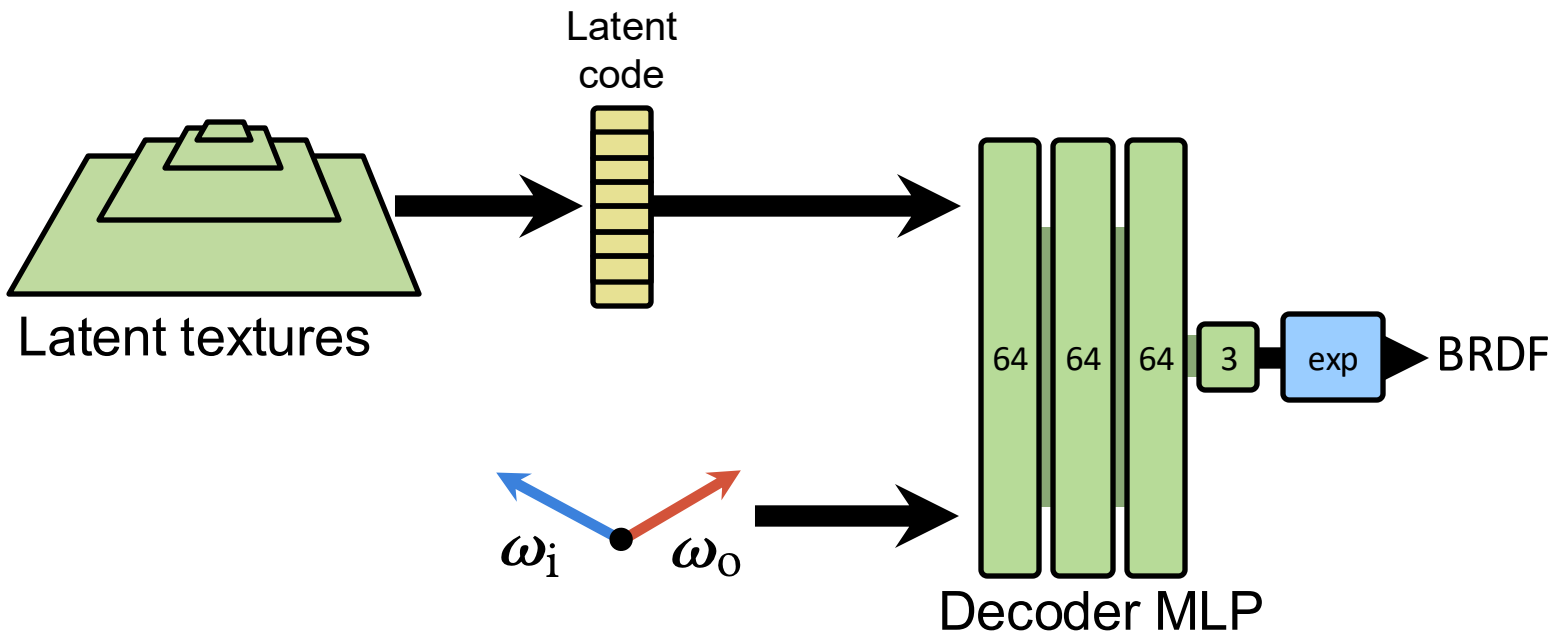



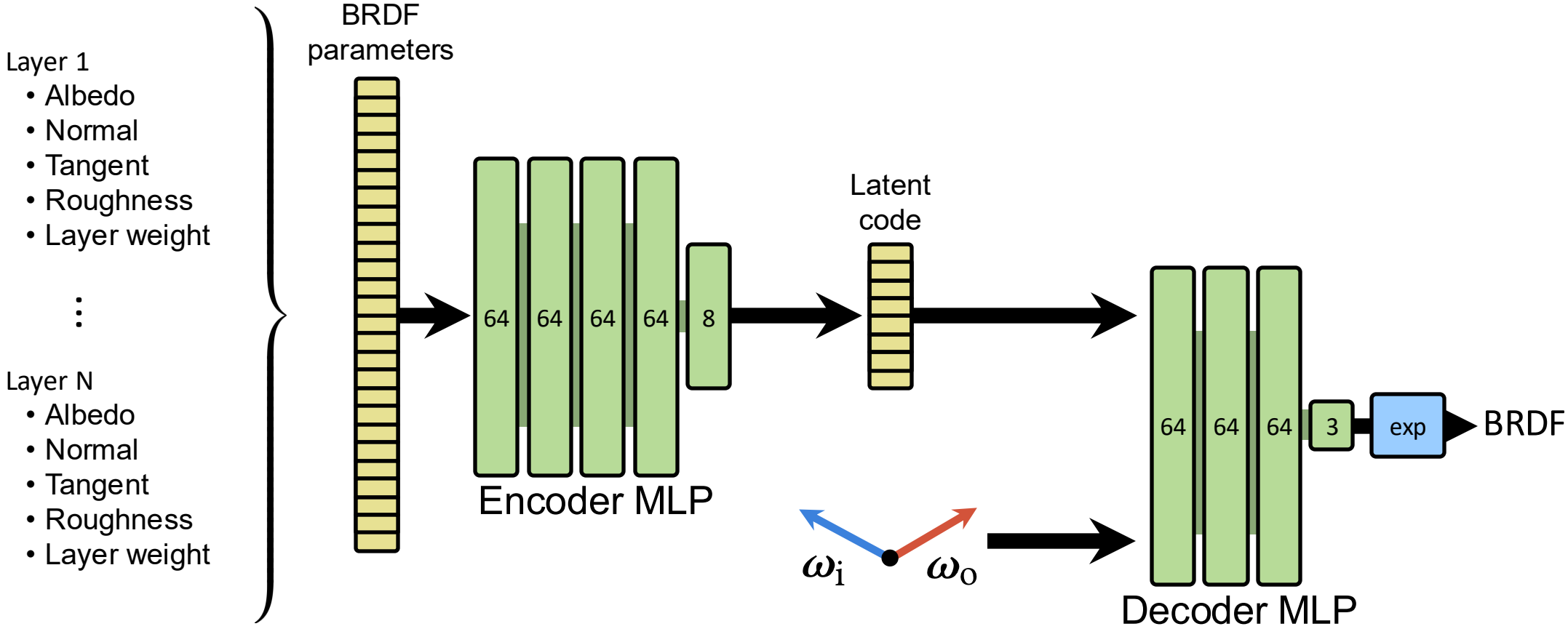
# WE'RE LACKING A "GLOBAL" VIEW





# IDEA: LEARN TO TRANSLATE THE ORIGINAL TEXTURES



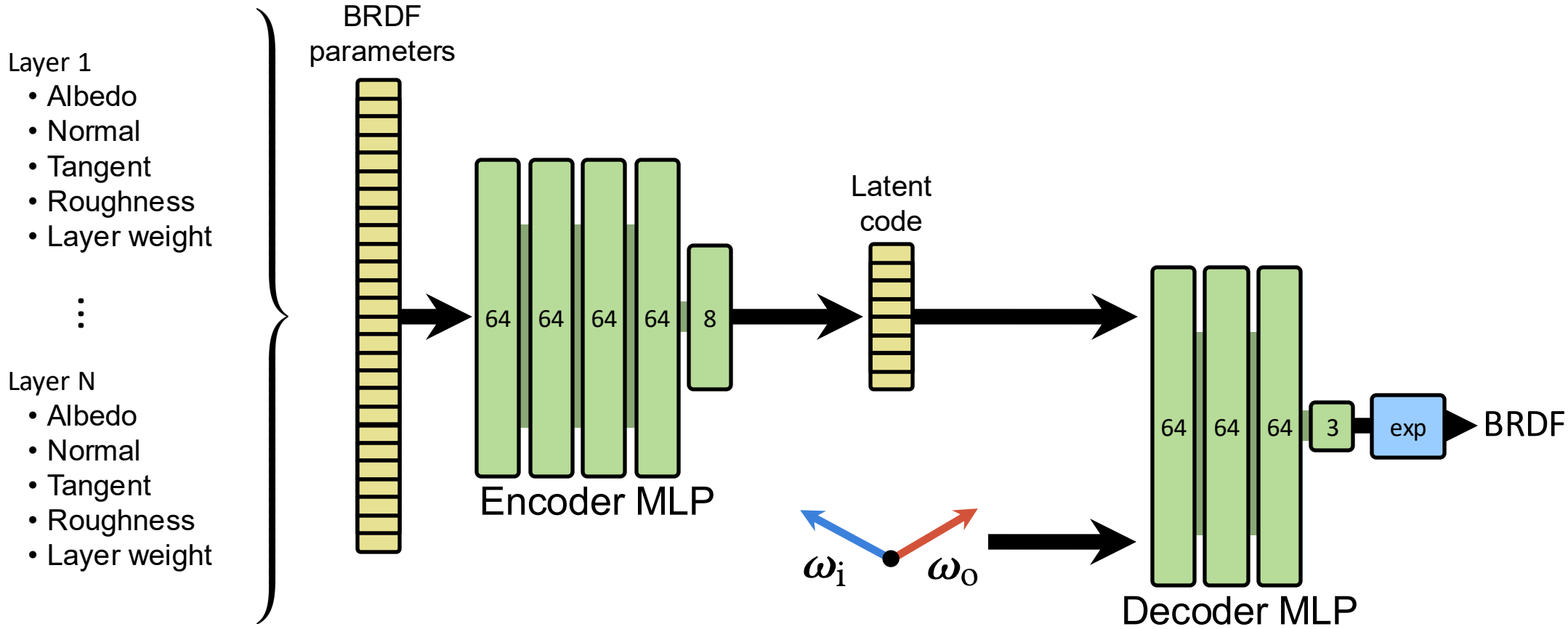


# IDEA: LEARN TO TRANSLATE THE ORIGINAL TEXTURES



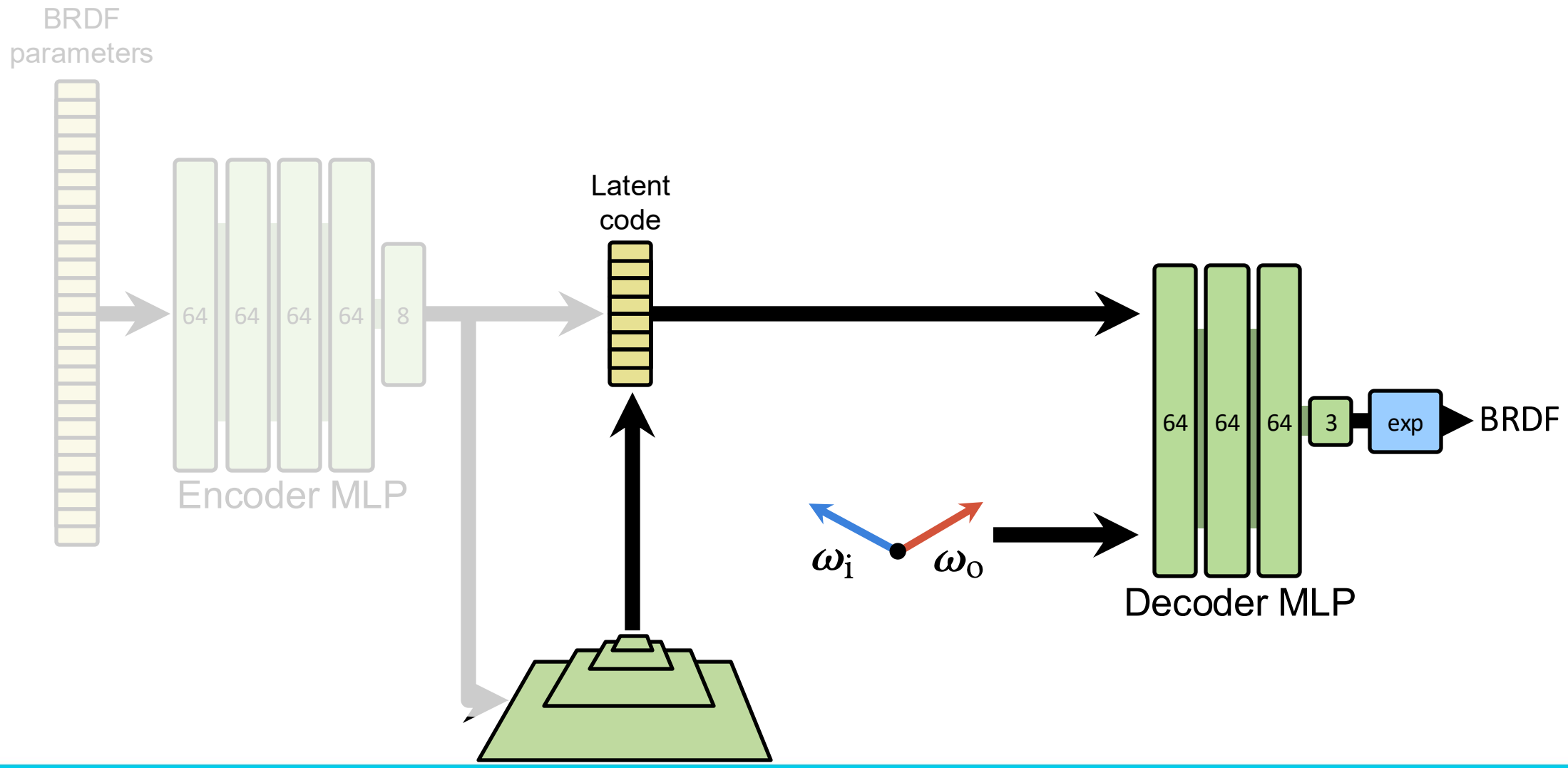
# YOU ONLY NEED THIS TRICK DURING TRAINING!

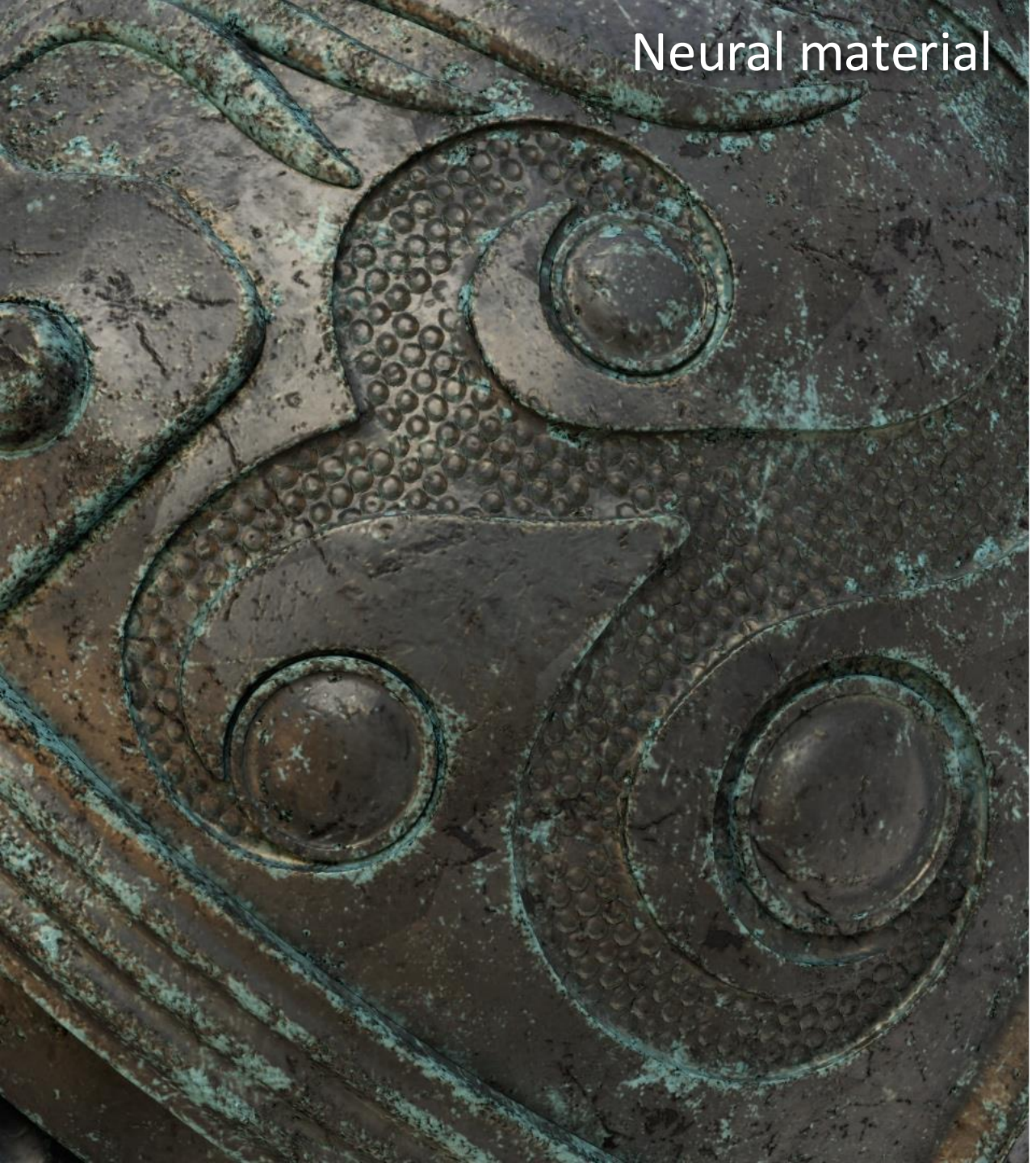


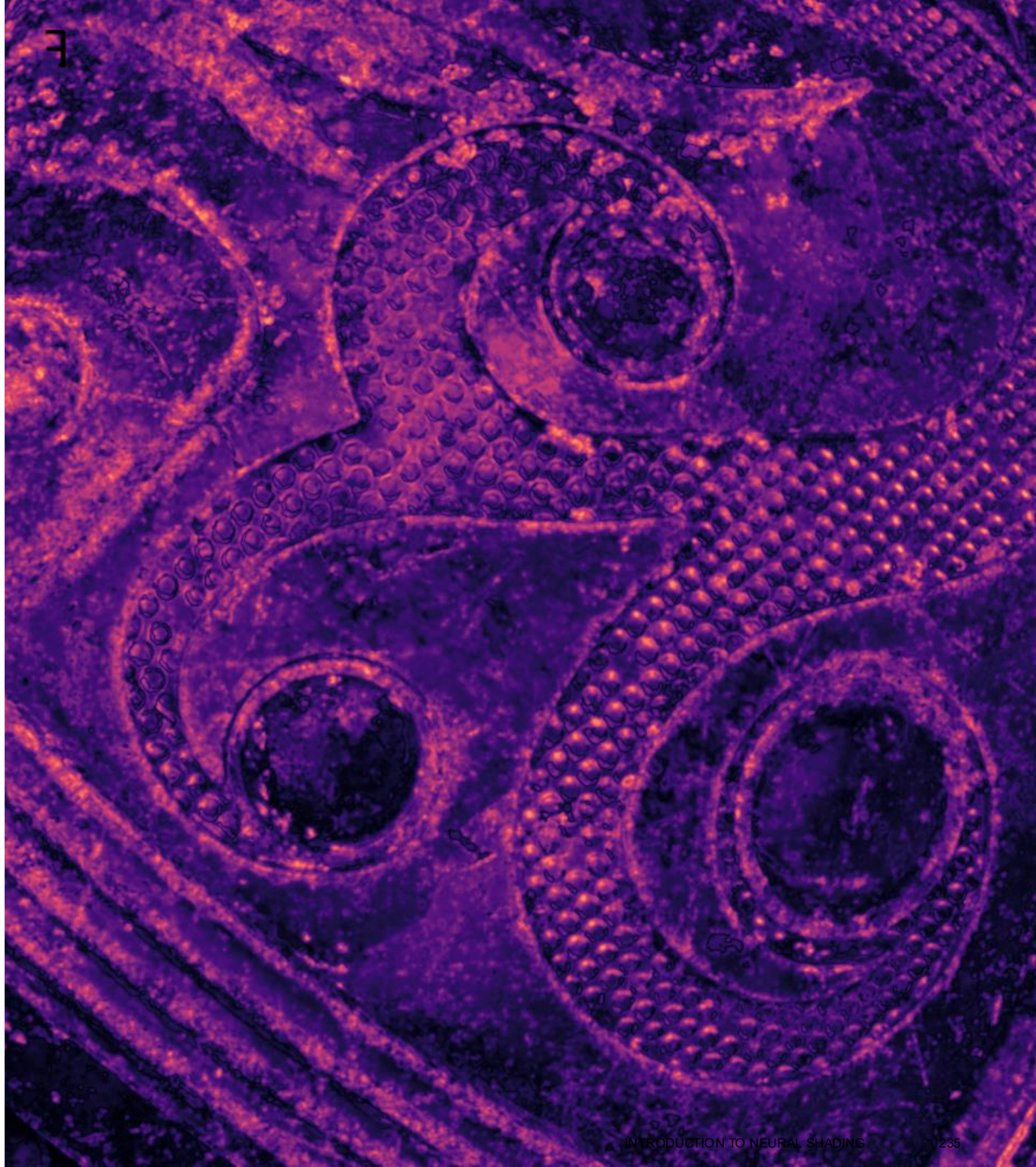


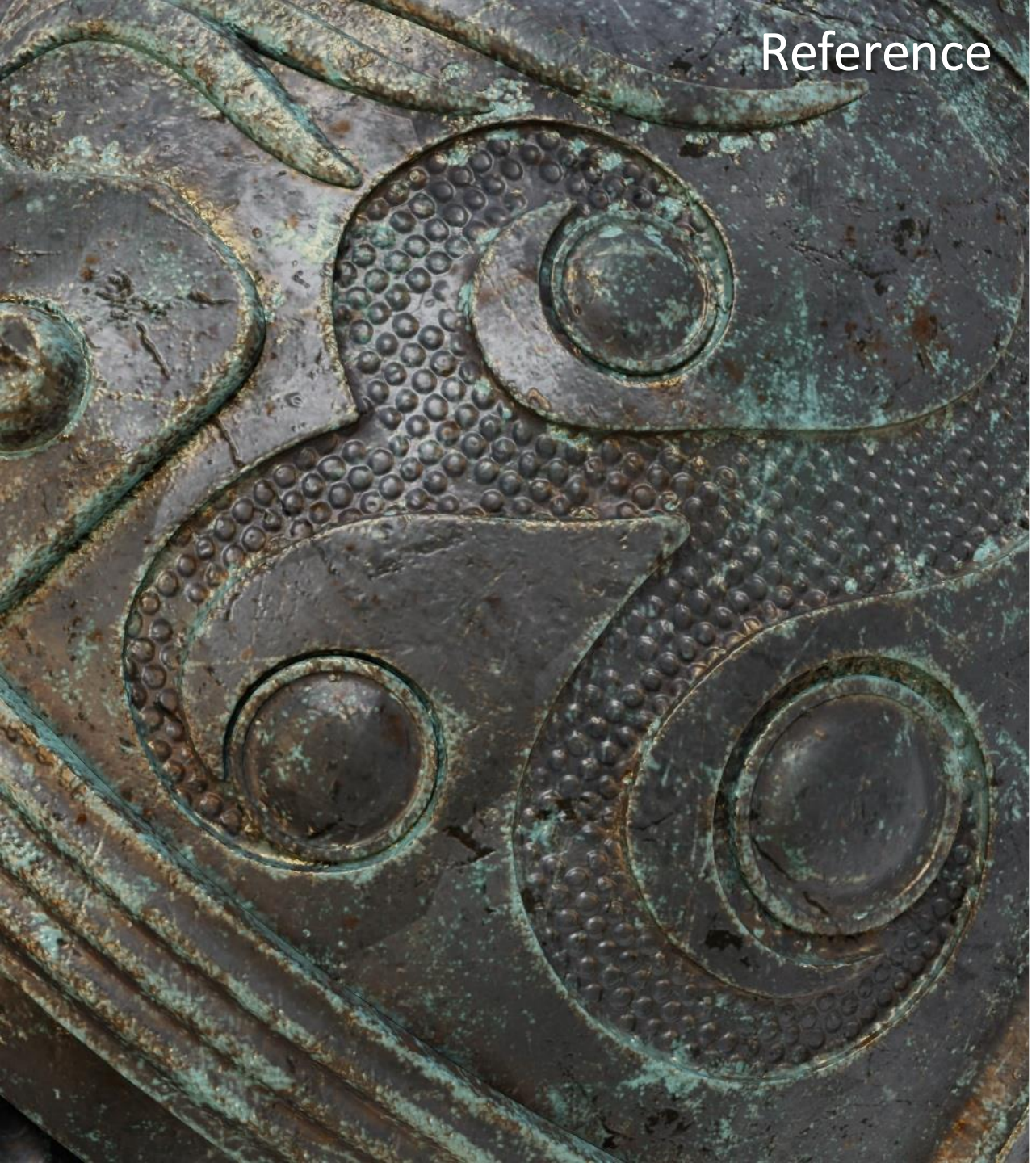
# YOU ONLY NEED THIS TRICK DURING TRAINING!

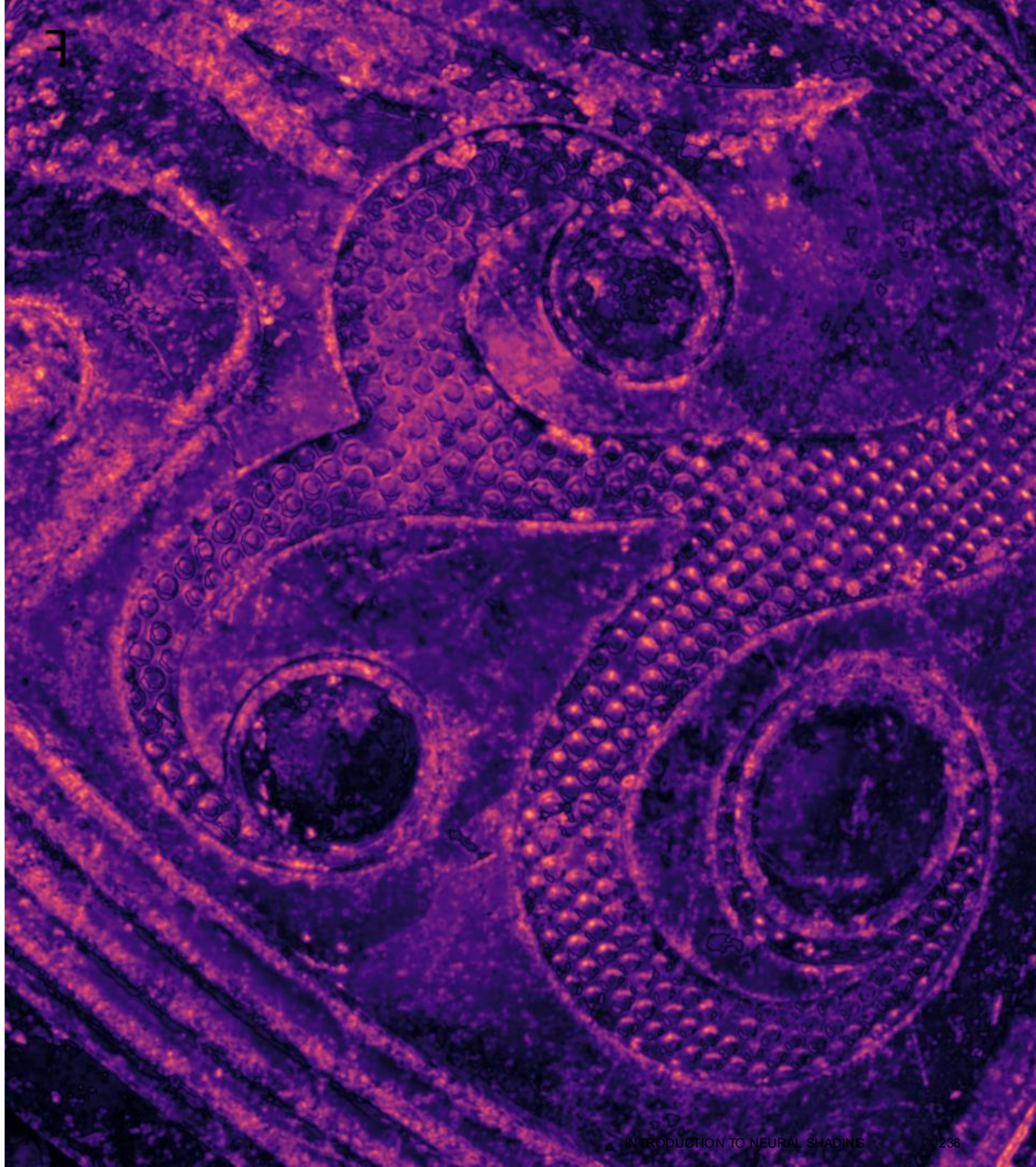





Neural material

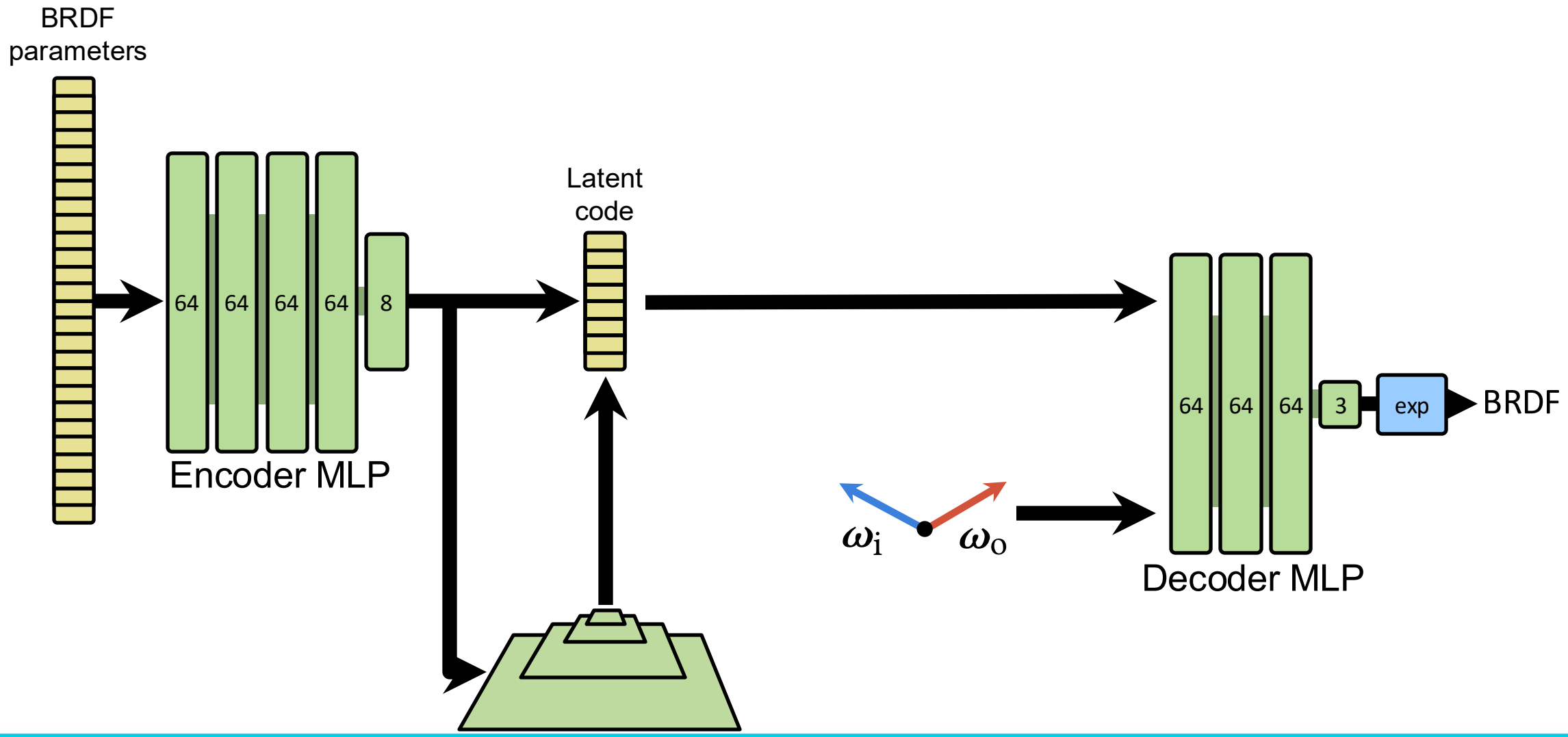



Reference



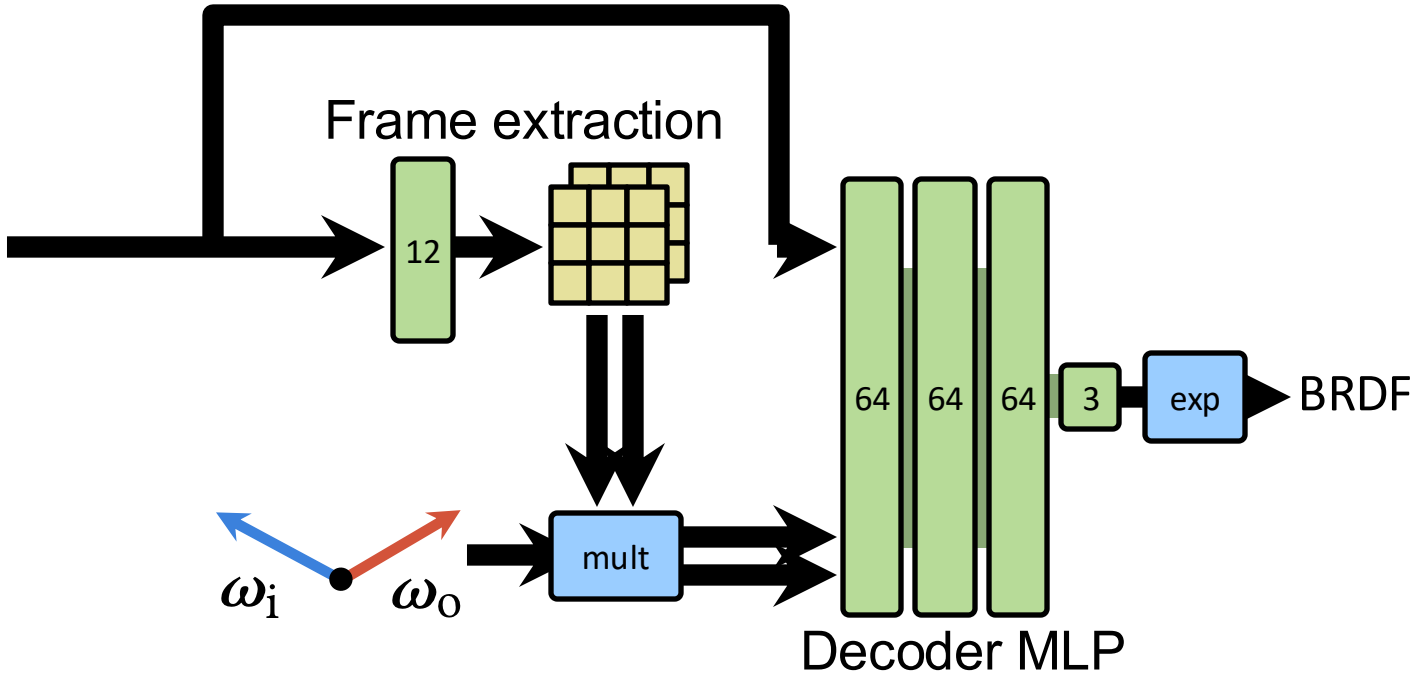
# LET'S ADD A FIXED-FUNCTION FRAME TRANSFORM





# LET'S ADD A FIXED-FUNCTION FRAME TRANSFORM




Neural without rotations

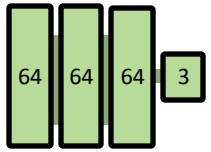
# Neural with rotations


Reference

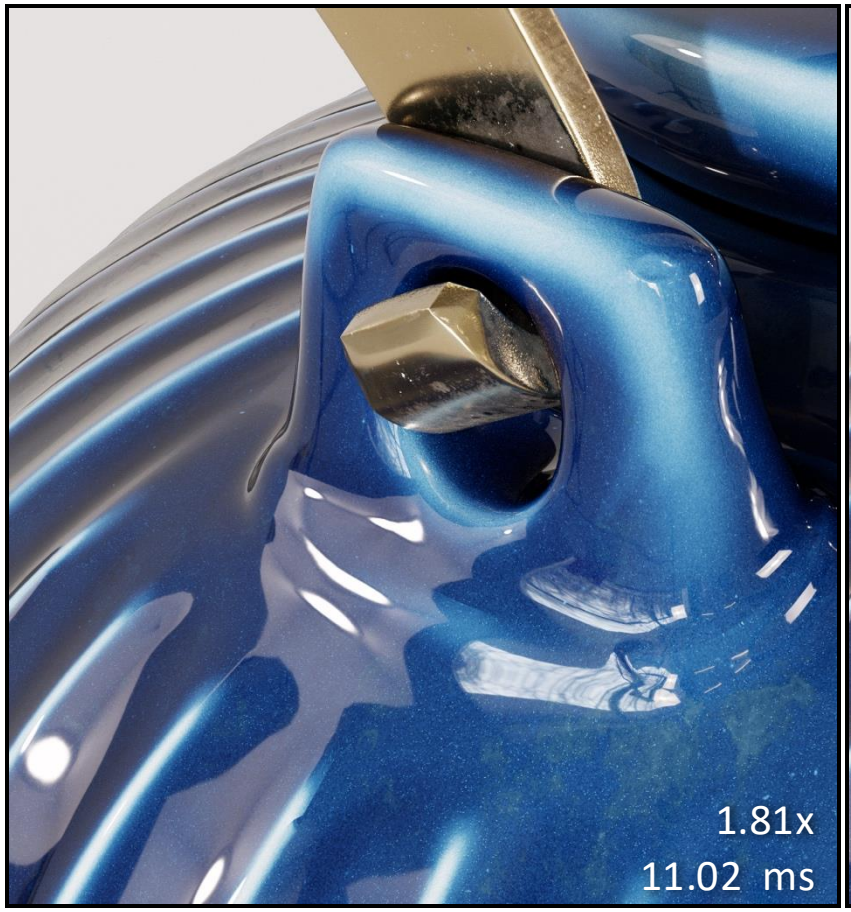
# END RESULT



Neural
3x64 wide layers



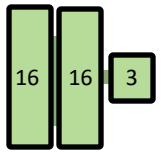

Reference




# NEURAL SHADING GIVES YOU A QUALITY/COST DIAL

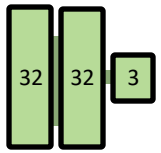


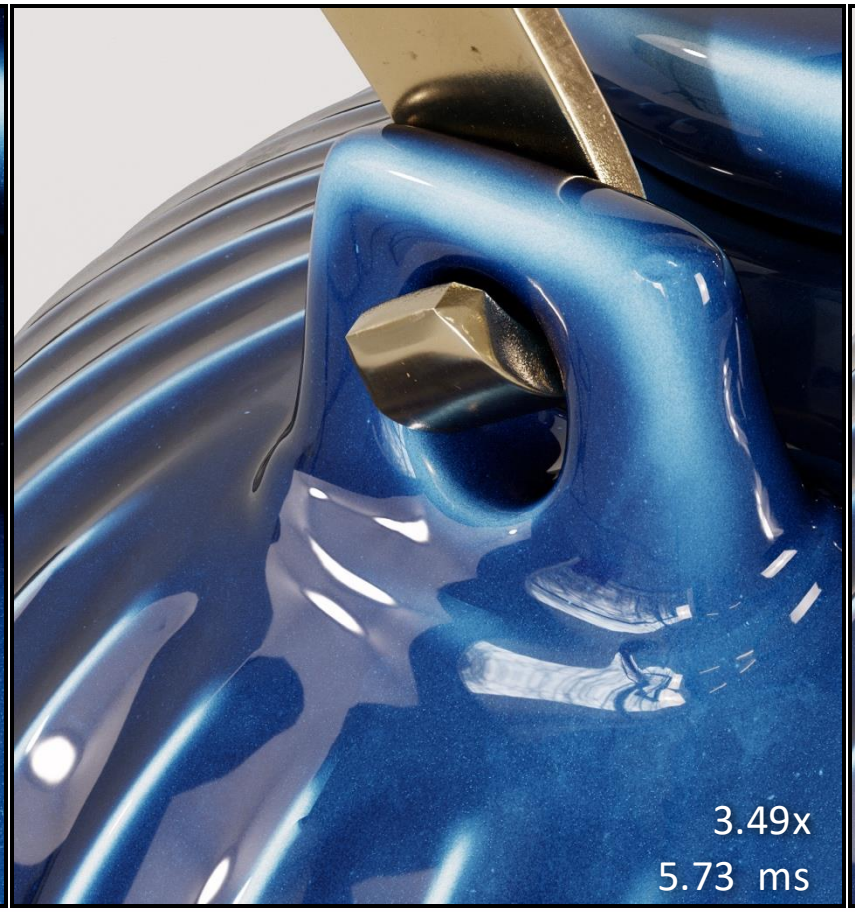
Neural
2x16 wide layers



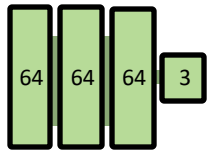


Neural
2x32 wide layers

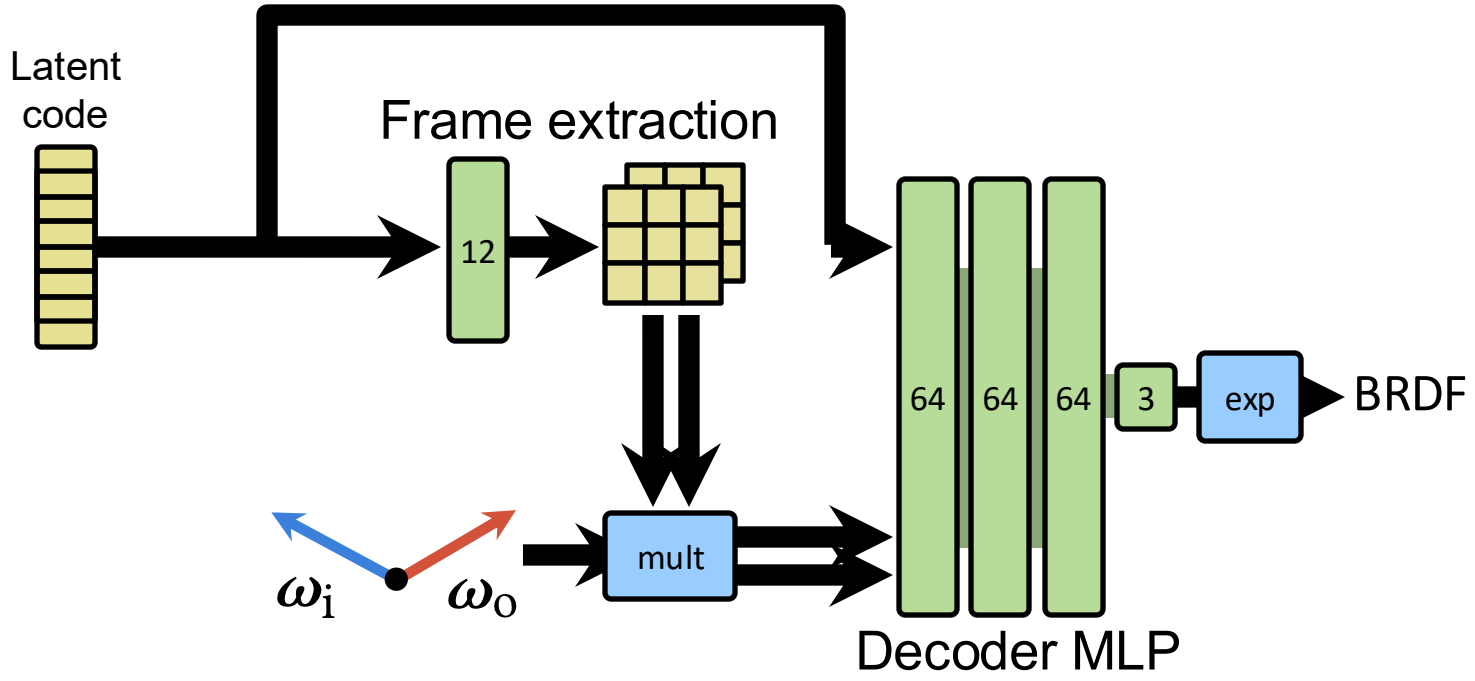



Neural
3x64 wide layers

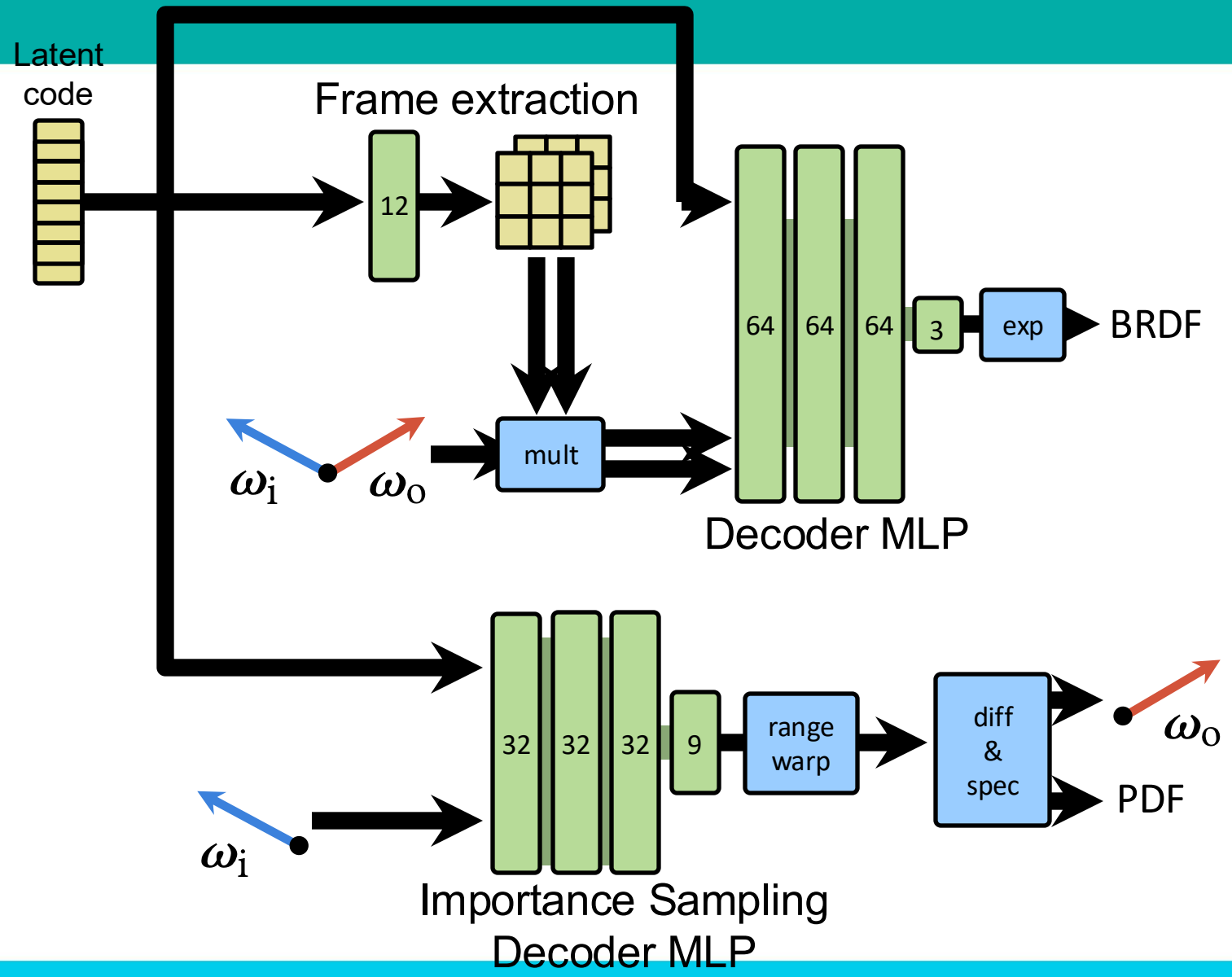



# ADDING ADDITIONAL CAPABILITIES



# ADDING ADDITIONAL CAPABILITIES





# MATERIAL LOD

# TAKE-AWAYS



- Some problems that are difficult to solve with classical shaders…
  - e.g. shader simplification, appearance filtering, …
- …become much easier when your shaders become trainable
- “Show it the error it’s making, and try to drive that lower”

But:

- You need new (debugging) skills
  - e.g. clever fixed-function components to off-load work from the network
- You need flexible code you can iterate over quickly
- Slang and SlangPy are designed to help you do that - give it a try!

SIGGRAPH 2025
Vancouver+ 10-14 August

# WRAP-UP

# THANK YOU!

SIGGRAPH 2025
Vancouver+ 10-14 August

- Course materials: https://shader-slang.org/landing/siggraph-25
- Related sessions:
  - **Hands-On Class: Introduction to Slang: The Next Generation Shading Language**
    - Sunday, 4pm – 5:30pm, West Building, Rooms 121-122
  - **Nsight Graphics Gaussian Splatting Sessions**
    - Wednesday, 9am – 10:30am & 1pm – 2:55pm, Room 116/117
  - **Birds of a Feather: Developing with Slang: Tools, Techniques, and Future Directions**
    - Wednesday, 2:30pm – 3:30pm, Vancouver Marriot Pinnacle Hotel
  - **Hands-on Vulkan® Ray Tracing With Dynamic Rendering**
    - Thursday, 11:45am – 1:15pm, West Building, Exhibit Hall B

# Q&A