Vulkanised 2025

The 7th Vulkan Developer Conference
Cambridge, UK | February 11-13, 2025

Using Neural Networks in Shaders

Jeff Bolz, NVIDIA

Vulkanised 2025

The 7th Vulkan Developer Conference
Cambridge, UK | February 11-13, 2025

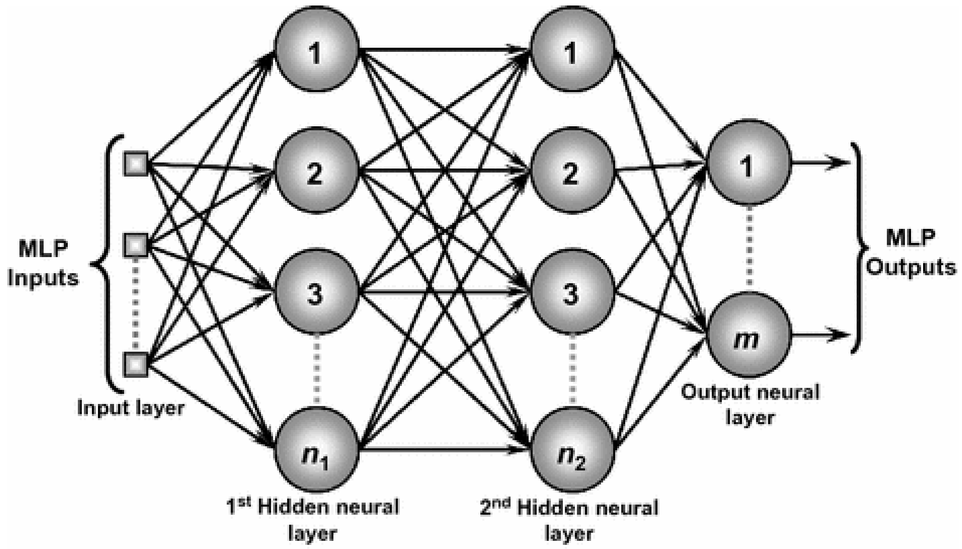
Using Neural Networks in Shaders

Jeff Bolz, NVIDIA

Background: What is an MLP?

Multi-Layer Perceptron aka Fully-Connected Network

- Each neuron is a function of all neurons in prior layer
  - Size is often described as “LxN”, where L is the number of hidden layers and N is number of neurons per layer
- Computing each layer involves:
  - **A matrix-vector multiply (e.g. NxN * Nx1)**
  - Optionally, add a bias vector
  - Activation function (e.g. ReLU(x) = max(x,0))
- *Evaluate the entire MLP in each thread*
  - Sizes like 2x16 up to 3x64 will be common



da Silva, et al, Multilayer Perceptron Networks

Neural Texture Compression (NTC)

https://research.nvidia.com/labs/rtr/neural_texture_compression/

Random-Access Neural Compression of Material Textures

KARTHIK VAIDYANATHAN*, NVIDIA, USA
MARCO SALVI*, NVIDIA, USA
BARTLOMIEJ WRONSKI*, NVIDIA, USA
TOMAS AKENINE-MÖLLER, NVIDIA, Sweden
PONTUS EBELIN, NVIDIA, Sweden
AARON LEFOHN, NVIDIA, USA

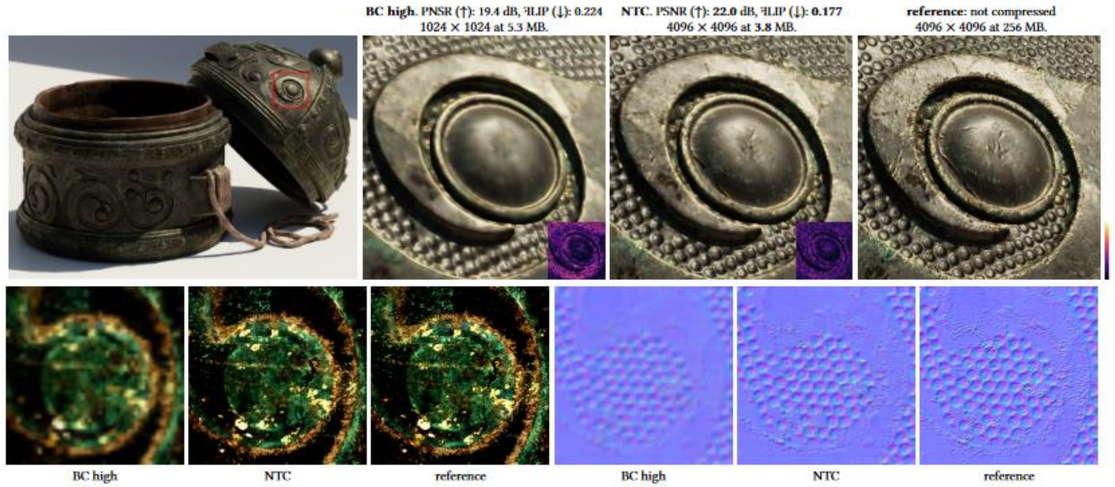
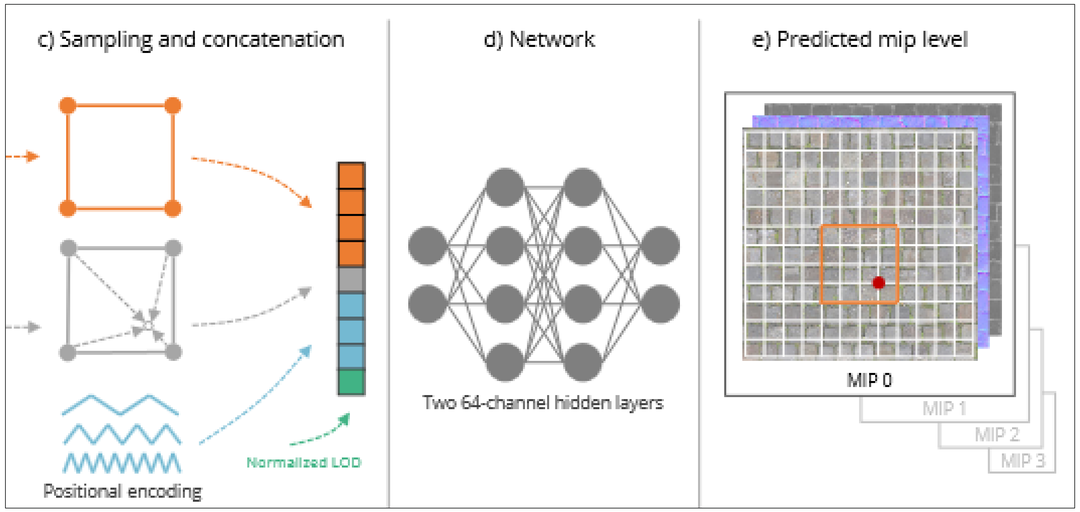


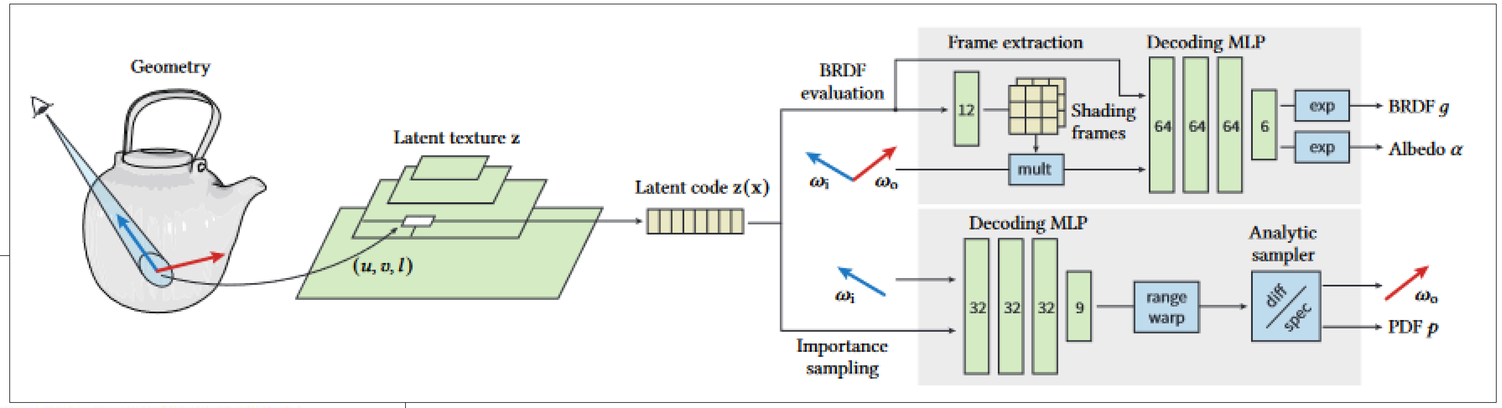
Fig. 1. A rendered image of an inkwell. The cutouts demonstrate quality using, from left to right, GPU-based texture formats (BC high) at 1024 × 1024 resolution, our neural texture compression (NTC), and high-quality reference textures. Note that NTC provides a 4× higher resolution (16× texels) than BC high, despite using 30% less memory. The PSNR and ꟻLIP quality metrics, computed for the cutouts, are shown above the respective images. The ꟻLIP error images are shown in the lower right corners, where brightness is proportional to error. Bottom row: two of the textures that were used for the renderings.



2x64 MLP

Neural Materials

https://research.nvidia.com/labs/rtr/neural_appearance_models/



Real-Time Neural Appearance Models

TIZIAN ZELTNER*, FABRICE ROUSSELLE*, ANDREA WEIDLICH*, PETRIK CLARBERG*, JAN NOVÁK*, BENEDIKT BITTERLI*, ALEX EVANS, TOMÁŠ DAVIDOVIČ, SIMON KALLWEIT, and AARON LEFOHN,
NVIDIA, Global

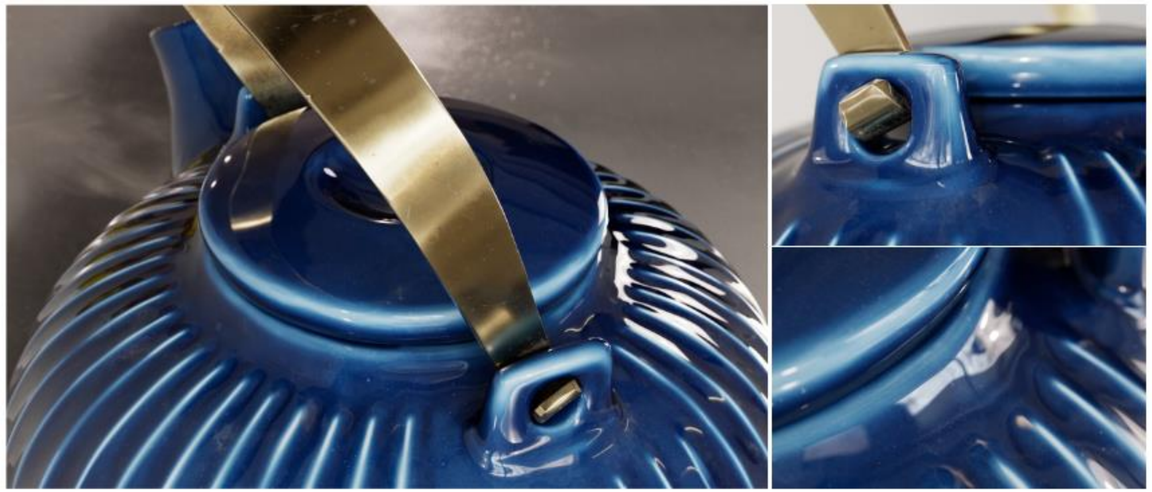


Fig. 1. Close-up renderings of a TEAPOT asset with our neural BRDF. Our model learns the intricate details and complex multi-layered material behavior of the ceramic, fingerprints, smudges, and dust which are responsible for the realism of the object while being faster to evaluate than traditional non-neural models of similar complexity. The system we present allows us to include such high-fidelity objects in real-time renderers in a scalable way.

3x64 + 3x32 MLPs


30.95 FPS
32.31 ms
Press [Left/Right Arrow] to cycle cameras
Press [SpaceBar] to orbit

Neural Intersection

https://gpuopen.com/download/publications/HPG2023_NeuralIntersectionFunction.pdf

Neural Intersection Function

S. Fujieda C. C. Kao T. Harada

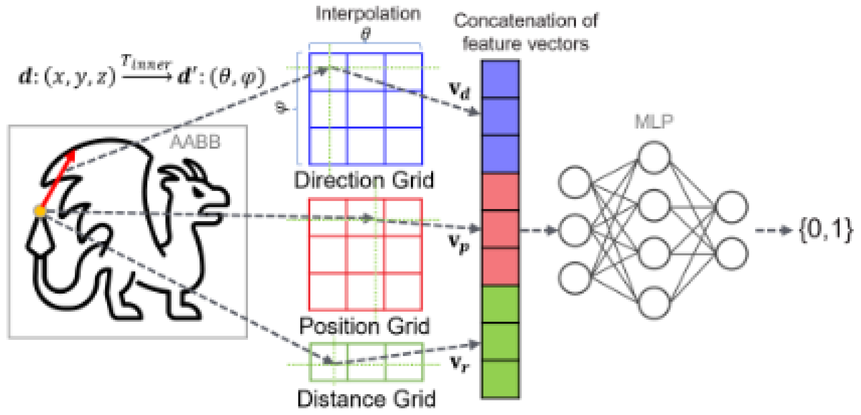
Advanced Micro Devices, Inc.

(a) (b) (c)

Figure 1: (a) A rendered image using Neural Intersection Function after 64 training samples per pixel. (b) A rendered image using ray tracing with BVH. (c) Difference ×3 between (a) and (b). PSNR is 39.11 dB. The scene has 30M triangles rendered at 1920 × 1080 on AMD Radeon™ RX 7900 XT. Secondary ray casting times are 4.54 ms and 5.27 ms in (a) and (b), respectively.

Abstract

The ray casting operation in the Monte Carlo ray tracing algorithm usually adopts a bounding volume hierarchy (BVH) to accelerate the process of finding intersections to evaluate visibility. However, its characteristics are irregular, with divergence in memory access and branch execution, so it cannot achieve maximum efficiency on GPUs. This paper proposes a novel Neural Intersection Function based on a multilayer perceptron whose core operation contains only dense matrix multiplication with predictable memory access. Our method is the first solution integrating the neural network-based approach and BVH-based ray tracing pipeline into one unified rendering framework. We can evaluate the visibility and occlusion of secondary rays without traversing the most irregular and time-consuming part of the BVH and thus accelerate ray casting. The experiments show the proposed method can reduce the secondary ray casting time for direct illumination by up to 35% compared to a BVH-based implementation and still preserve the image quality.



(b) NIF inner network

2x64 + 3x48 MLPs

NVIDIA.

MobileNeRF

https://mobile-nerf.github.io/

MobileNeRF: Exploiting the Polygon Rasterization Pipeline for Efficient Neural Field Rendering on Mobile Architectures

Zhiqin Chen[1,2,4] Thomas Funkhouser[1] Peter Hedman[1] Andrea Tagliasacchi[1,2,3,4]
Google Research[1] Simon Fraser University[2] University of Toronto[3]

Abstract

Neural Radiance Fields (NeRFs) have demonstrated amazing ability to synthesize images of 3D scenes from novel views. However, they rely upon specialized volumetric rendering algorithms based on ray marching that are mismatched to the capabilities of widely deployed graphics hardware. This paper introduces a new NeRF representation based on textured polygons that can synthesize novel images efficiently with standard rendering pipelines. The NeRF is represented as a set of polygons with textures representing binary opacities and feature vectors. Traditional rendering of the polygons with a z-buffer yields an image with features at every pixel, which are interpreted by a small, view-dependent MLP running in a fragment shader to produce a final pixel color. This approach enables NeRFs to be rendered with the traditional polygon rasterization pipeline, which provides massive pixel-level parallelism, achieving interactive frame rates on a wide range of compute platforms, including mobile phones.

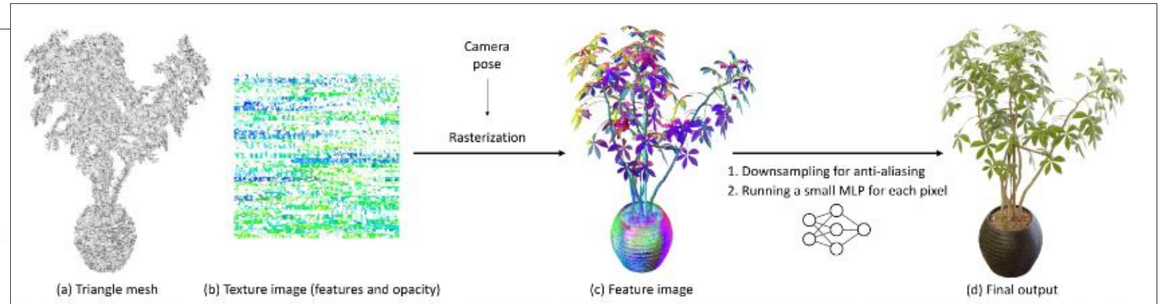
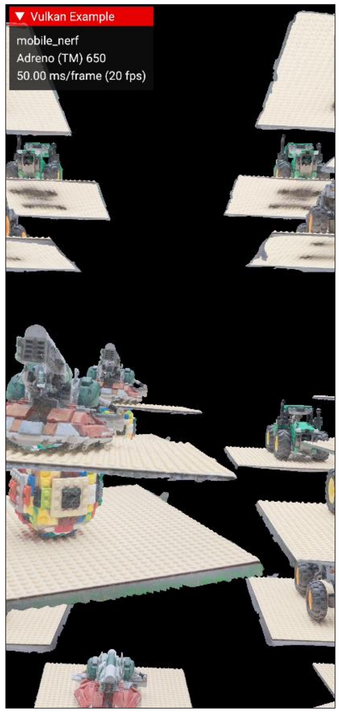


Figure 2. **Overview (rendering)** – We represent the scene as a triangle mesh textured by deep features. We first rasterize the mesh to a deferred rendering buffer. For each visible fragment, we execute a neural deferred shader that converts the feature and view direction to the corresponding output pixel color.

Figure 1. **Teaser** – We present a NeRF that can run on a variety of common devices at interactive frame rates.

algorithm that evaluates a large MLP at hundreds of sample positions along the ray for each pixel in order to estimate and integrate density and radiance. This rendering process




```
for (int j = 0; j < NUM_CHANNELS_ONE; ++j) {
    if (intermediate_one[j] <= 0.0) {
        continue;
    }
    for (int i = 0; i < NUM_CHANNELS_TWO; ++i) {
        intermediate_two[i] += intermediate_one[j] *
            texelFetch(weightsOne, ivec2(j, i), 0).x;
    }
}
```

Actual MobileNeRF code

2x16 MLP

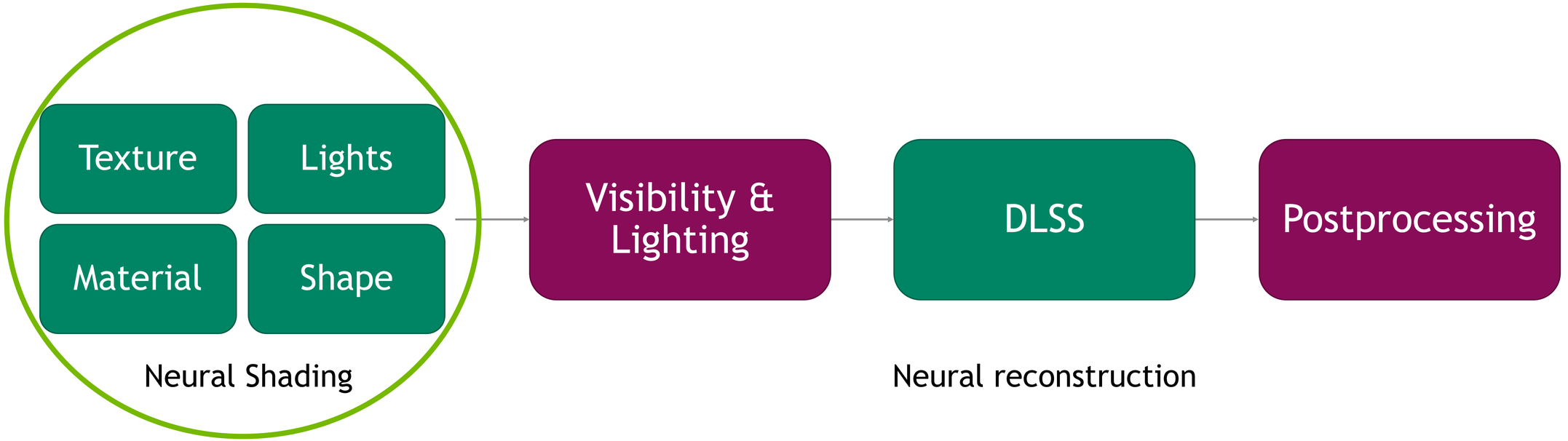
https://github.com/KhronosGroup/Vulkan-Samples/pull/849

NVIDIA

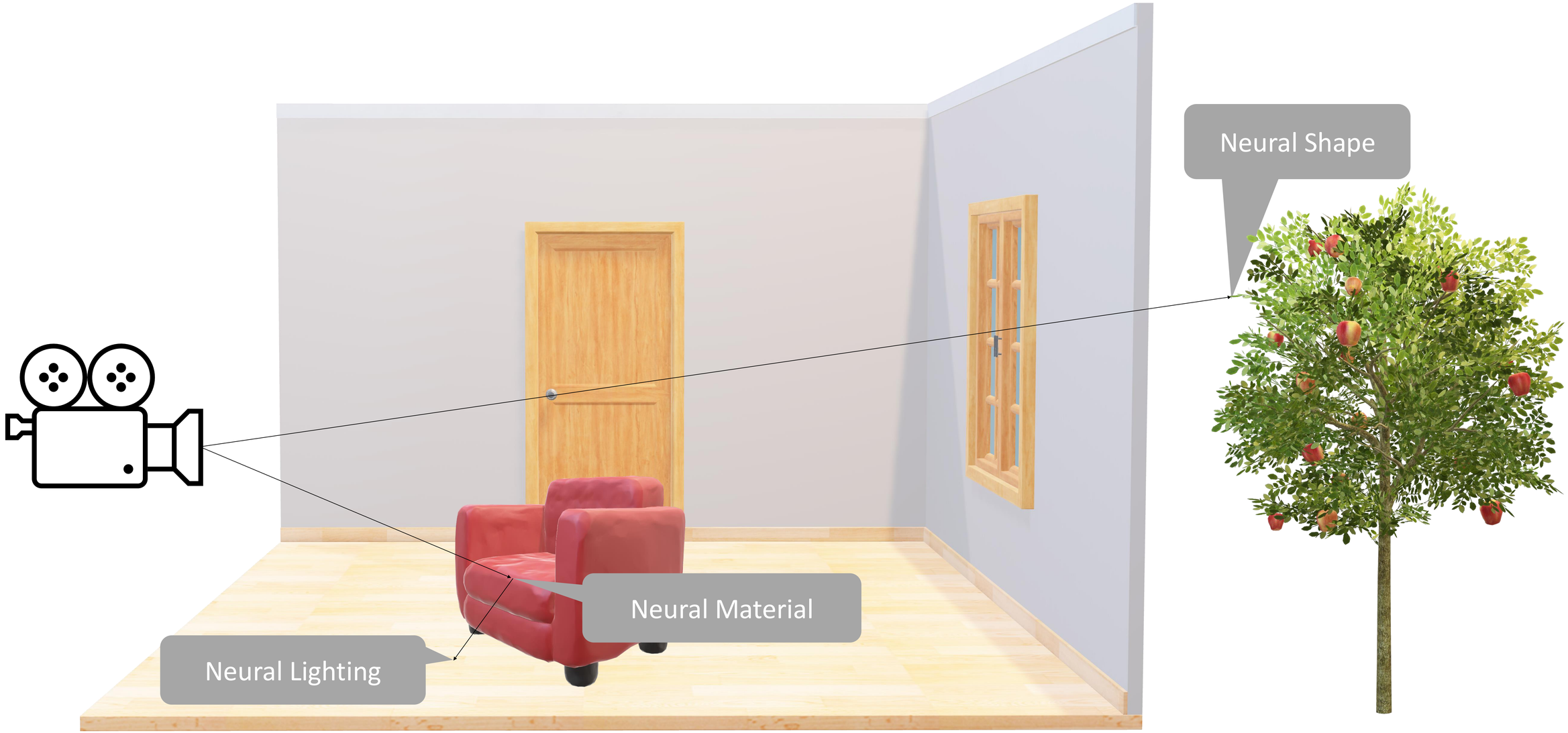
Neural Shading

Evaluate MLP per-ray or per-pixel

- Research has shown:
  1. These tiny networks *work* – they faithfully reproduce the effect they're trained on
  2. They're *small* enough to fit in a shader
  3. They're *fast* enough to run in real-time – often more efficient than physically-based shading
- What's missing is shading language support to accelerate these!



Rendering a Scene with Neural Models



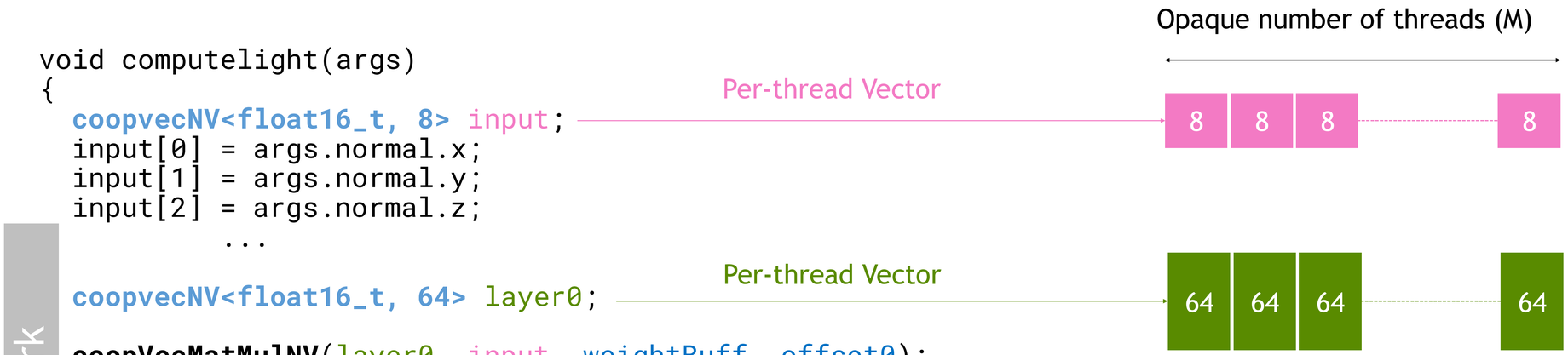
A Programming Model for Neural Shaders

Goals

- Accelerate neural shaders using a **thread-based/SIMT** programming model
  - Compatible with current shader programming models
  - Neural networks can be easily plugged into existing shaders
- Enable optimizations for neural shaders in compiler and hardware
  - Enabling tensor cores, divergence handling, layer fusion, etc.
  - Developers should have no need for baking-in hardware specific decisions
- Portability across current and future hardware
  - Can target a variety of matrix acceleration HW
  - Allow innovations for neural shaders in future hardware
- Supported in **every** shader stage - **with tensor cores!**

A Programming Model for Neural Shaders

Arbitrary-sized vector types



Neural Network

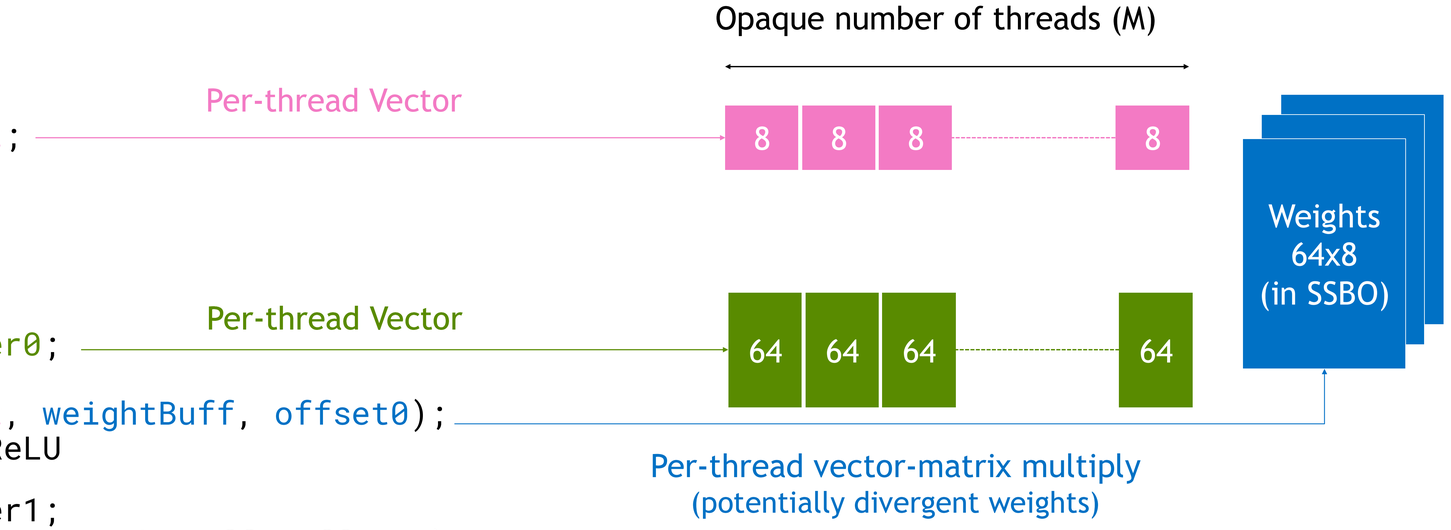
```
void computelight(args)
{
  coopvecNV<float16_t, 8> input;
  input[0] = args.normal.x;
  input[1] = args.normal.y;
  input[2] = args.normal.z;
           ...

  coopvecNV<float16_t, 64> layer0;

  coopVecMatMulNV(layer0, input, weightBuff, offset0);
  layer0 = max(layer0, 0); // ReLU

  coopvecNV<float16_t, 64> layer1;
  coopVecMatMulNV(layer1, layer0, weightBuff, offset1);
  layer1 = max(layer1, 0); // ReLU
  ...

  coopvecNV<float16_t, 4> output;
  coopVecMatMulNV(output, layerN, weightBuff, offsetN);
  output = exp(output);

  color.r = output[0] * args.lightColor;
  color.g = output[1] * args.lightColor;
  color.b = output[2] * args.lightColor;
          …
}
```

A Programming Model for Neural Shaders

Matrix-vector multiply intrinsics reference weight matrices in memory

Neural Network

```
void computelight(args)
{
  coopvecNV<float16_t, 8> input;
  input[0] = args.normal.x;
  input[1] = args.normal.y;
  input[2] = args.normal.z;
           ...

  coopvecNV<float16_t, 64> layer0;

  coopVecMatMulNV(layer0, input, weightBuff, offset0);
  layer0 = max(layer0, 0); // ReLU

  coopvecNV<float16_t, 64> layer1;
  coopVecMatMulNV(layer1, layer0, weightBuff, offset1);
  layer1 = max(layer1, 0); // ReLU
  ...

  coopvecNV<float16_t, 4> output;
  coopVecMatMulNV(output, layerN, weightBuff, offsetN);
  output = exp(output);

  color.r = output[0] * args.lightColor;
  color.g = output[1] * args.lightColor;
  color.b = output[2] * args.lightColor;
          …
}
```

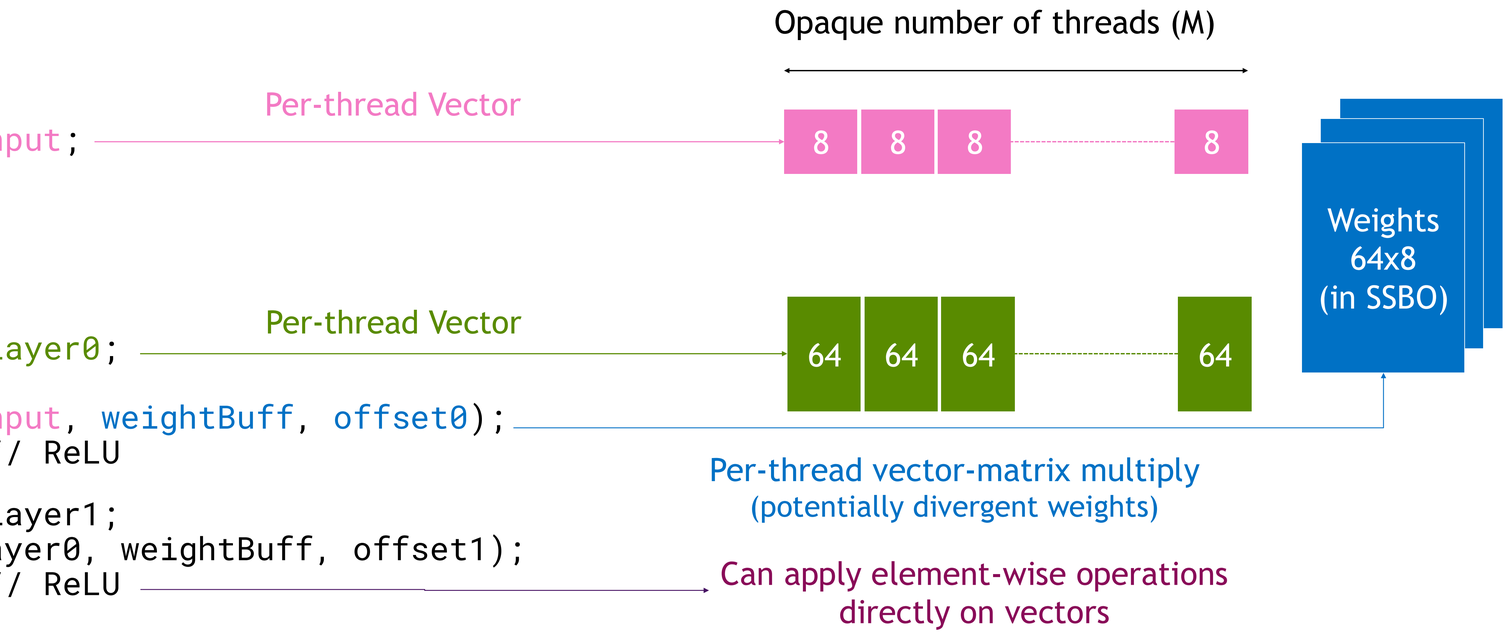


A Programming Model for Neural Shaders

Can apply element-wise operations to opaque vectors

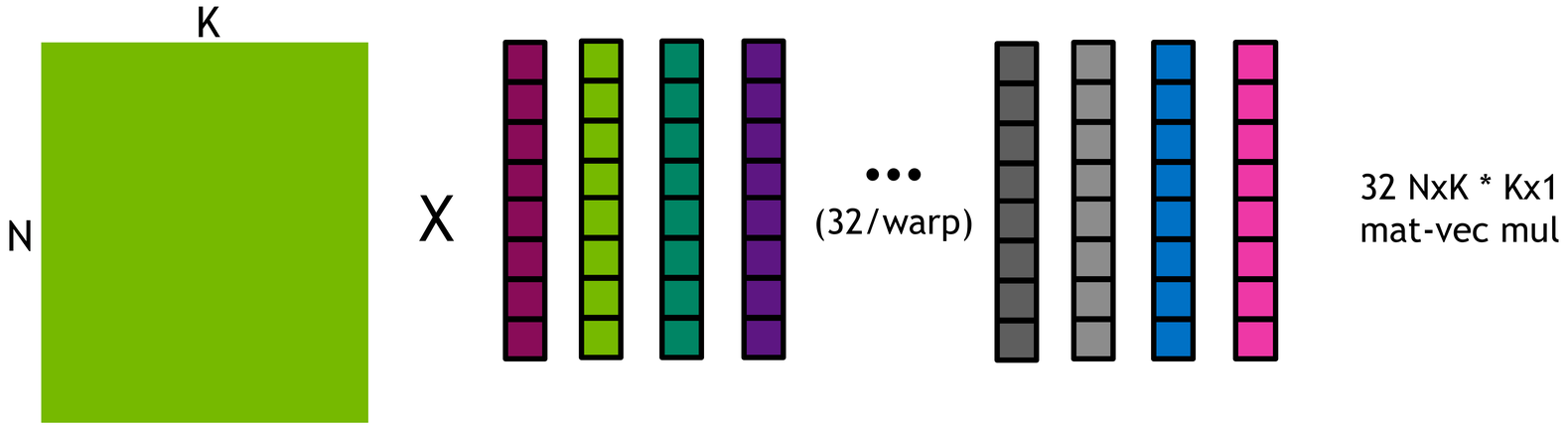
Neural Network

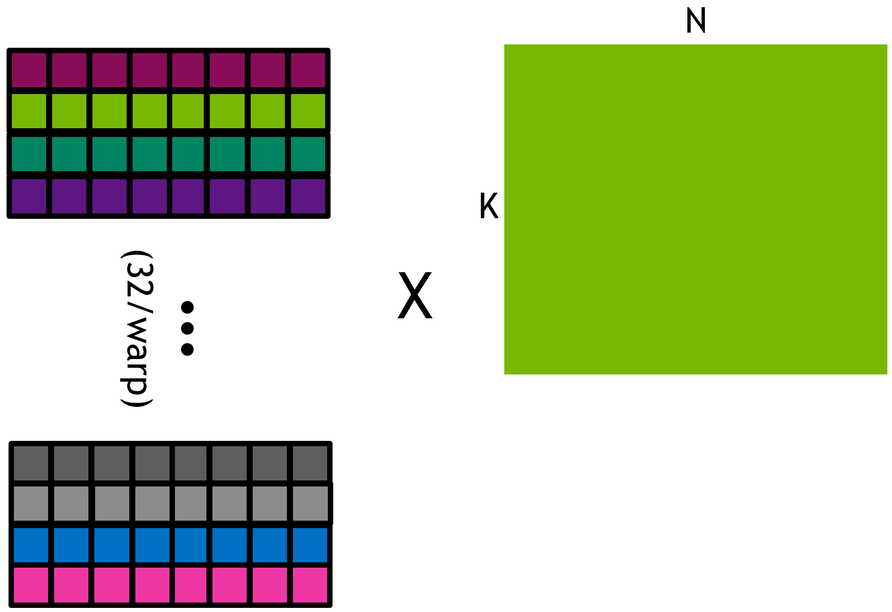
```
void computelight(args)
{
  coopvecNV<float16_t, 8> input;
  input[0] = args.normal.x;
  input[1] = args.normal.y;
  input[2] = args.normal.z;
           ...

  coopvecNV<float16_t, 64> layer0;

  coopVecMatMulNV(layer0, input, weightBuff, offset0);
  layer0 = max(layer0, 0); // ReLU

  coopvecNV<float16_t, 64> layer1;
  coopVecMatMulNV(layer1, layer0, weightBuff, offset1);
  layer1 = max(layer1, 0); // ReLU
  ...

  coopvecNV<float16_t, 4> output;
  coopVecMatMulNV(output, layerN, weightBuff, offsetN);
  output = exp(output);

  color.r = output[0] * args.lightColor;
  color.g = output[1] * args.lightColor;
  color.b = output[2] * args.lightColor;
          …
}
```



Vectors as A-Matrix

An implementation detail





Combine to one
NxK * Kx32 mat-mat mul

Internally, implemented
as 32xK * KxN mat-mat mul,
tensor core accelerated

Cooperative Vectors

Implementation aspects

- New templated vector types
  - Arbitrary dimension, but known at compile-time
  - They *opportunistically cooperate* behind the scenes when performing matrix-vector multiply
  - May transparently switch between “SIMT” and “cooperative” layouts as needed, but programming model is always SIMT

- Compiler can optimize across sequences of vector ops
  - Fusion of back-to-back MMA ops - minimize switching between layouts
  - Individual element access can break fusion – prefer whole-vector ops

- Support common unary and binary element-wise ops
  - Activation functions: ReLU, Tanh, etc. standard binary ops: +, -, *, /

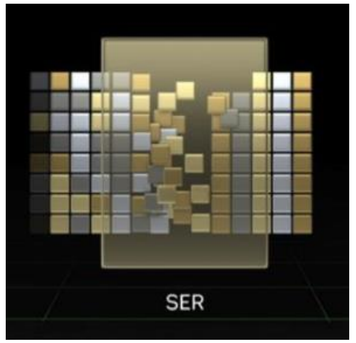
Divergence and Disabled Threads

Implementation aspects

- Matmul behavior well defined under control flow divergence and nonuniformity
  - Each thread sees the result of its own matrix-vector multiply

- Implementation is responsible for handling control flow divergence and nonuniformity
  - Peeling loop?
  - Fall-back to per-thread math?
  - Something else?
  - Performance is expected to degrade gracefully as divergence increases

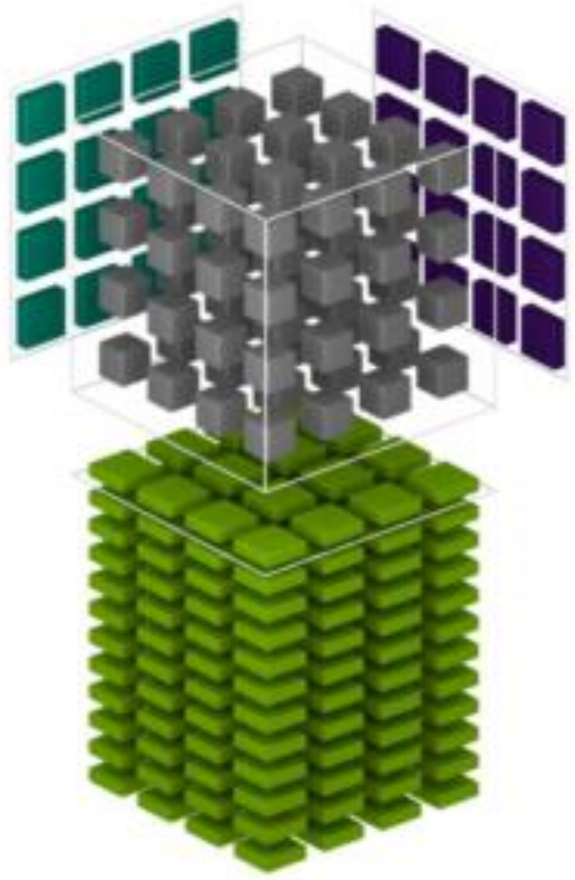
- Coherence *not required* functionally, but still *important* for performance
  - Composable with VK_EXT_ray_tracing_invocation_reorder
    - Sort threads by network



Contrast vs Cooperative Matrix

- CoopMat: Access to tensor core through compute shaders
  - Explicitly defined to operate at group scope
  - Uses a fixed set of power-of-two matrix shapes
  - Targets kernel-scope MMAs
  - Shader responsible for staging matrix loads from memory
  - **Results undefined under thread divergence**

- CoopVec: Neural shaders need a “SIMT-friendly” version of MMA
  - **Support for shader stages w/o explicit concurrency**
  - Arbitrary matrix and vector sizes - IHV independent
  - Can optimize matrix loads and divergence handling
  - Support for fusion across layers



GLSL Extension Details

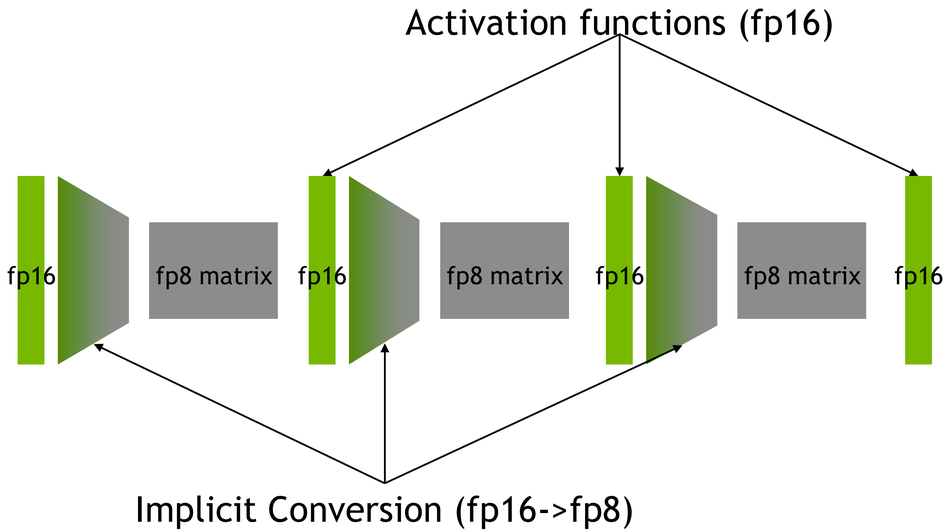
GL_NV_cooperative_vector

```
void coopVecMatMulAddNV(
    out coopvecNV<ResultTy, ResultComps> result,        // output vector
    coopvecNV<InputTy, InputComps> input,               // input vector
    uint inputInterpretation,                           // type to convert input to, before matmul
    const MatrixTy[] matrix,                            // matrix array in SSBO memory
    uint matrixOffset,                                  // byte offset where matrix starts
    uint matrixInterpretation,                          // type of matrix data
    const BiasTy[] bias,                                // bias array in SSBO memory (added to matmul result)
    uint biasOffset,                                    // byte offset where bias starts
    uint biasInterpretation,                            // type of bias data
    uint M,                                             // number of elements in result vector
    uint K,                                             // number of element in input vector
    uint matrixLayout,                                  // row-major, col-major, optimal
    bool transpose,                                     // transpose the matrix
    uint matrixStride);                                 // stride for row/col-major matrices
```

“Interpretation” Parameters

Serve multiple purposes

- Specify type of weight/bias values in memory
- Allows type conversion of the input vector
  - E.g. fp16 -> fp8 without real fp8 type
  - “Fast path” type conversion and layout change
- Allow bitcast of the input vector for small types
  - E.g. <N x i8> as <N/4 x i32>
  - No need for native fp8 or sub-8b types in the shading language



Extension Details

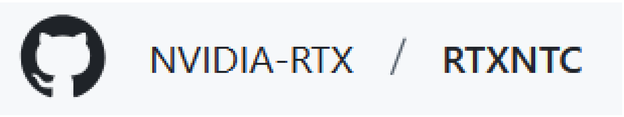
VK_NV_cooperative_vector

- Vulkan API commands to convert matrix to “optimal” layout
  - Host and device commands, meant to be used at load time
  - Pre-shuffles matrix so the shader can do optimal memory access
  - Pass in e.g. row-major layout, it writes out optimal layout
  - Critical for best performance
  - No need for a separate VkTensor resource, store in SSBO/ByteAddressBuffer
- Vulkan API command to query supported type combinations (similar to coop matrix)

inputType	inputInterpretation	matrixInterpretation	biasInterpretation	resultType
FP16	FP16	FP16	FP16	FP16
FP16	E4M3	E4M3	FP16	FP16
FP16	E5M2	E5M2	FP16	FP16
SINT8	SINT8	SINT8	SINT32	SINT32
UINT32	SINT8_PACKED	SINT8	SINT32	SINT32
FP32	SINT8	SINT8	SINT32	SINT32

Training

- How will game engines train the networks?
  - Some use cases like NTC can have a standalone tool



  - Some may be trained in PyTorch/SlangPy

  - Some may need to be trained in-engine (Slang w/autodiff)

- Two additional intrinsics help with training in-engine:
  - Outer-product accumulate
  - Vector accumulate

Backpropagation

New Training Intrinsics

- Biases are adjusted by derivative of activation function
  - Vector atomic reduction:

```
void coopVecReduceSumAccumulateNV(const coopvecNV<T, N> v,
                                  T[] buf, uint offset);
```

- Weights are adjusted by outer product
  - Outer product atomic reduction:

```
void coopVecOuterProductAccumulateNV(const coopvecNV<T, M> v1,
                                     const coopvecNV<T, N> v2,
                                     T[] buf, uint offset,
                                     int matrixLayout);
```

https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \qquad chain\ rule$$

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \qquad by\ definition$$

$$m\ -\ number\ of\ neurons\ in\ l-1\ layer$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \qquad by\ differentiation\ (calculating\ derivative)$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}a_k^{l-1} \qquad final\ value$$

Equations for derivative of C in a single weight (w_jk)^l

Backpropagation

```
/***** Forward pass ****/
x0 = matvecmul(input, W0, b0)
x0_a = act(x0)

x1 = matvecmul(x0_a, W1, b1)
x1_a = act(x1)

x2 = matvecmul(x1_a, W2, b2)
x2_a = act(x2)

/***** Loss  ****/
loss = (x2_a - ref)^2
loss_grad = 2 * (x2_a - ref)
```

```
/***** Backward Pass  ****/
x2_g = act_backward(loss_grad, x2)
ReduceSumAccumulate(x2_g, bias2_grad)
OuterProductReduce(x2_g, x1_a, weight2_grad)
x1_g = matvecmul(x2_g, W2, transpose)

x1_g = act_backward(x1_g, x1)
ReduceSumAccumulate(x1_g, bias1_grad)
OuterProductReduce(x1_g, x0_a, weight1_grad)
x0_g = matvecmul(x1_g, W1, transpose)

x0_g = act_backward(x0_g, x0)
ReduceSumAccumulate(x0_g, bias0_grad)
OuterProductReduce(x0_g, input, weight0_grad)
```

Directed Test Performance

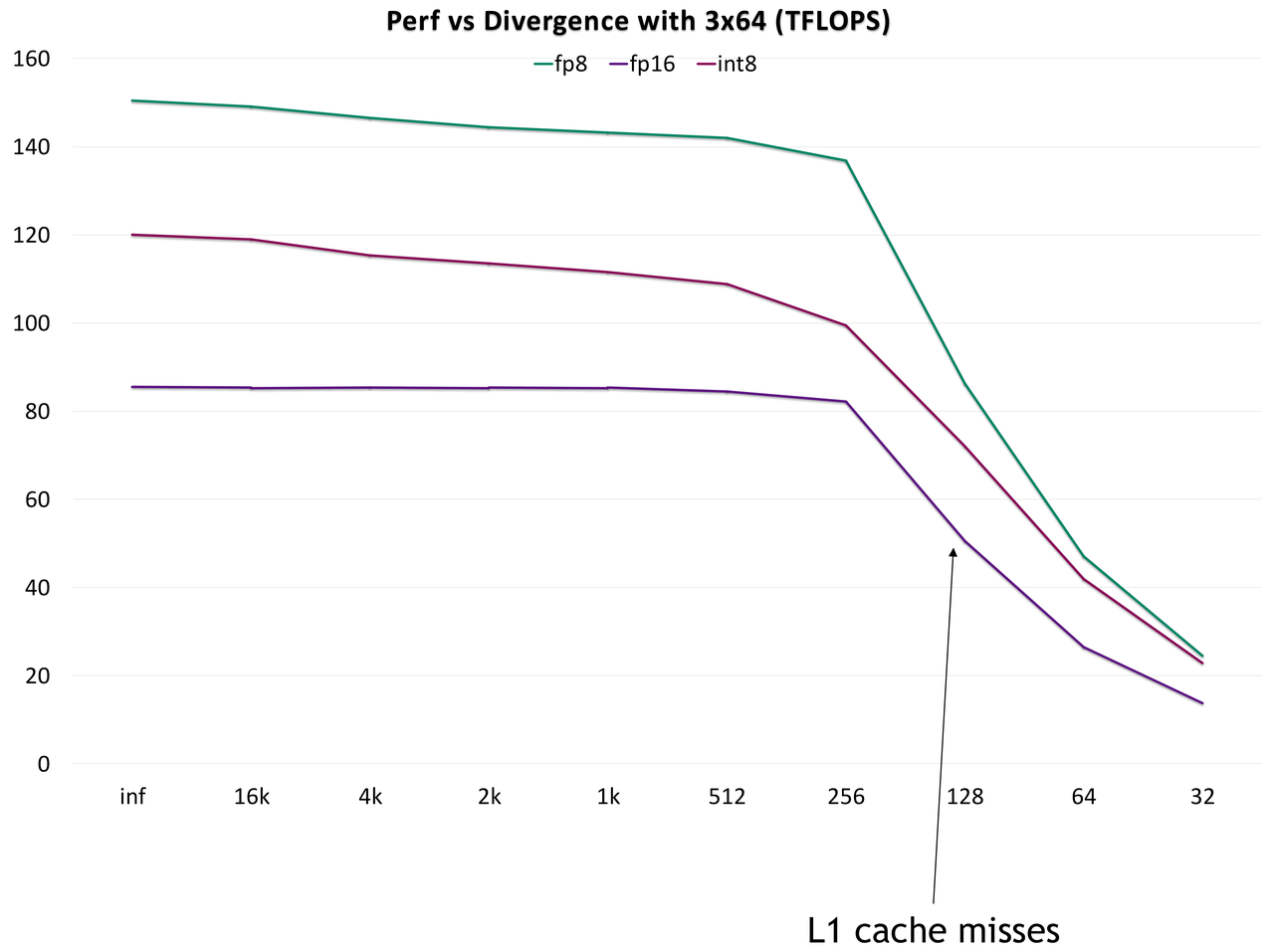
RTX 4070 (46 SMs @ 1920MHz)

Peak measured performance

TFLOPS	fp16	int8	fp8
2x32	73	43	85
3x32	79	59	121
2x64	84	102	141
3x64	85	119	150
SOL	90.5	181	181

Divergence (FP8, 3x64)

Threads/matrix	TFLOPS
inf	150
16k	149
4k	146
2k	144
1k	143
512	142
256	137
128	86
64	47
32	24

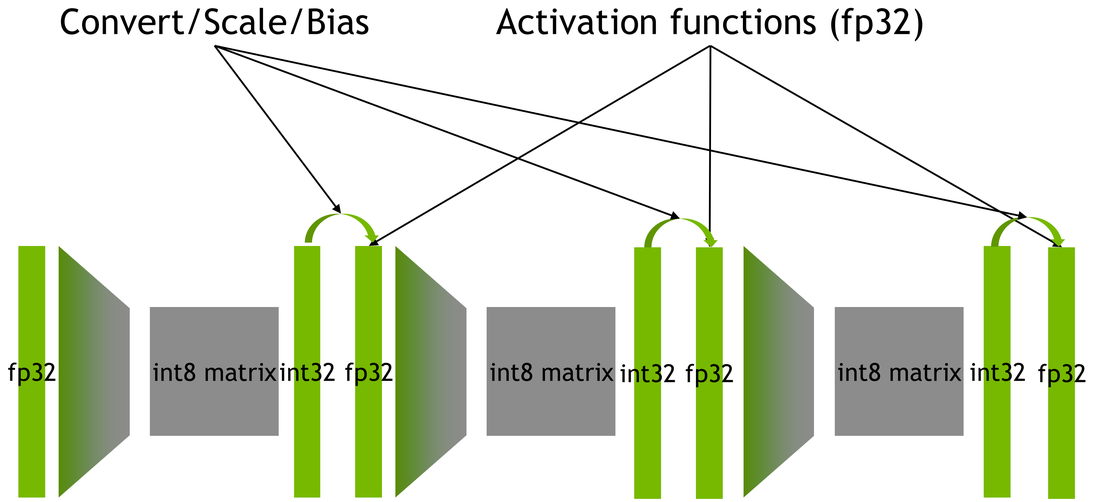


https://github.com/jeffbolznv/vk_cooperative_vector_perf/

INT8 vs FP8 Learnings

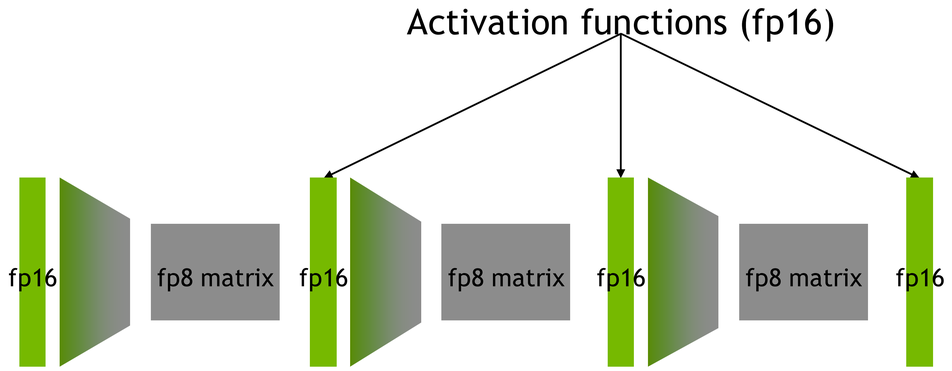
Conversions and activation functions are expensive

TFLOPS	fp16	int8	fp8
2x32	73	43	85
3x32	79	59	121
2x64	84	102	141
3x64	85	119	150
SOL	90.5	181	181



- INT8/FP8 2x peak rate of FP16, but INT8 can be *slower* than FP16
- For small K, activation/conversion costs dominate
- More complex activation functions will be slower

- FP8 support is **required** and is emulated as FP16 on older GPUs



Neural Texture Compression Performance

https://github.com/NVIDIA-RTX/RTXNTC

GPU	API	CoopVec	Time – View 1, ms	Time – View 2, ms
RTX 4090	DX12	No	6.95	6.05
RTX 4090	Vk	No	6.95	6
RTX 4090	DX12	Int8	1.45	1.32
RTX 4090	Vk	Int8	1.45	1.35
RTX 4090	DX12	FP8	1.28	1.2
RTX 4090	Vk	FP8	1.25	1.17
Intel A750	DX12	No	54.5	47
Intel A750	Vk	No	50.9	44.1
AMD RX6800XT	DX12	No	16.8	14.35
AMD RX6800XT	Vk	No	26.8	23.1

- Inference-on-sample
- CoopVec is 5x faster than DP4A inference

Conclusion

- Neural shading techniques moving from research to reality
  - Leveraging the speed of the tensor cores for graphics
- The Vulkan/SPIR-V/GLSL/Slang extensions are available now
  - Supported in R570 drivers
  - Supported on all NVIDIA RTX GPUs
  - Tooling available on github and will be in next Vulkan SDK
- DX support coming soon
  - https://devblogs.microsoft.com/directx/enabling-neural-rendering-in-directx-cooperative-vector-support-coming-soon/


NVIDIA