neural.slang

A Standard Module for Inline Neural Networks in Shaders

Kai Zhang

HPG 2026

What is Slang?

One modern shading language, compiled to every major GPU platform.

modern language features

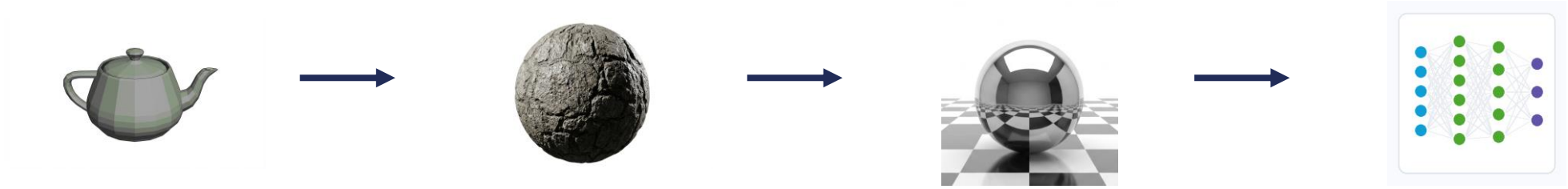
cross-platform by design

automatic differentiation

HPG 2026

Graphics compute keeps evolving

Graphics has always used better compute tools to build better approximations.



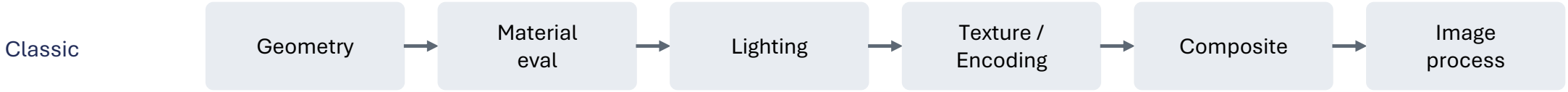
Fixed function	Programmable shaders	Ray tracing	Neural rendering
texture maps	BRDFs / materials	lighting paths	learned approximations

The latest step is letting neural networks learn some of the approximations for us.

HPG 2026

Neural compute moves inside the pipeline

Start from the rendering pipeline we already know.



Geometry, material evaluation, lighting, texture/encoding, compositing, and final image processing.

HPG 2026

Neural compute moves inside the pipeline

DLSS is the familiar case: neural compute after rendering.

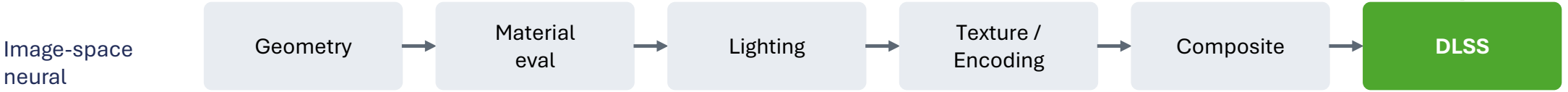


image in -> image out

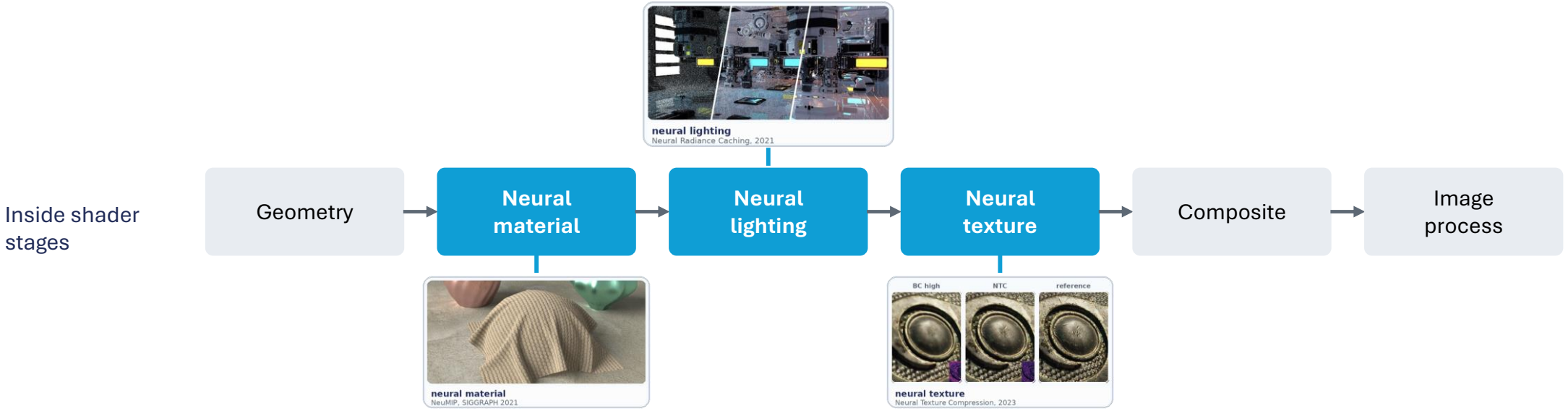
The neural network replaces the final image-processing box, while the middle of the pipeline remains traditional.

Image source: NVIDIA Research DLSS 4 / Ray Reconstruction

HPG 2026

Neural compute moves inside the pipeline

Neural compute can live inside shader stages.



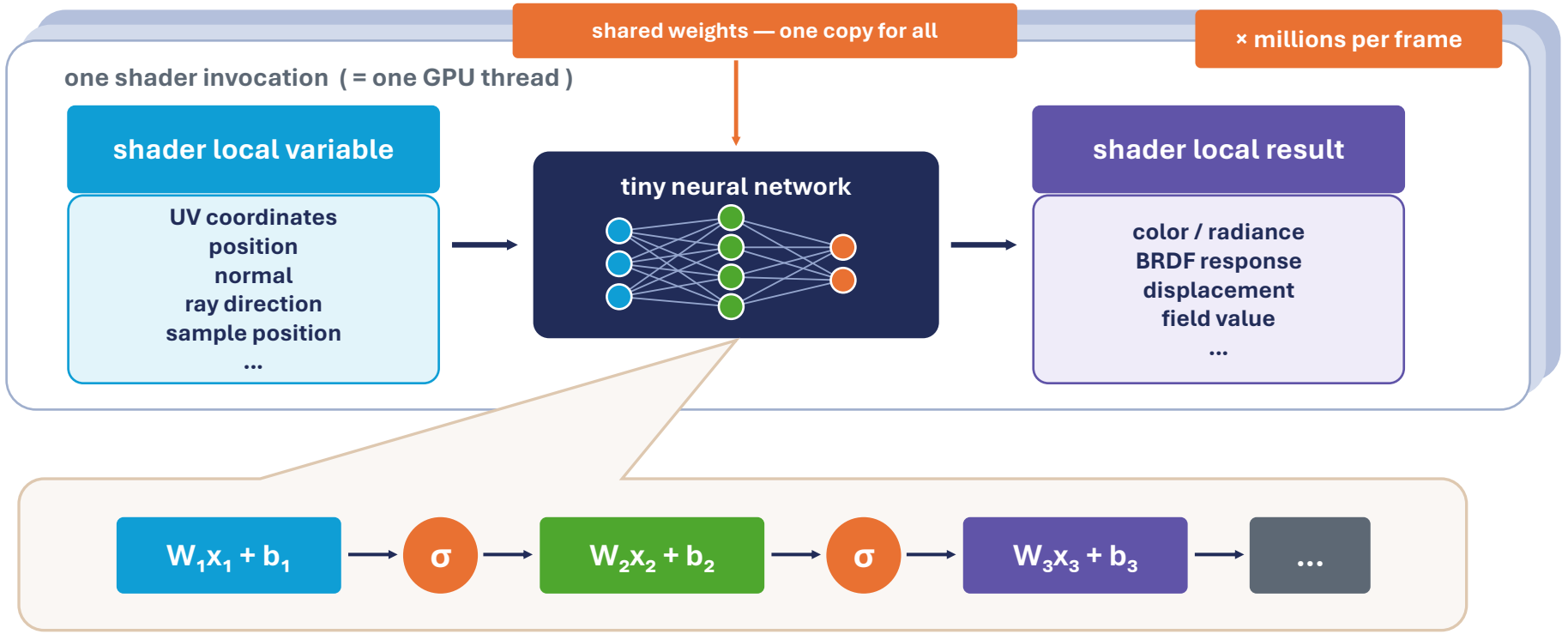
Same pipeline architecture; different compute inside selected boxes.

Image sources: NeuMIP (SIGGRAPH 2021), Neural Radiance Caching (2021), Random-Access Neural Compression of Material Textures (SIGGRAPH 2023)

HPG 2026

The neural shading workload

A neural building block is a tiny learned function inside the shader.

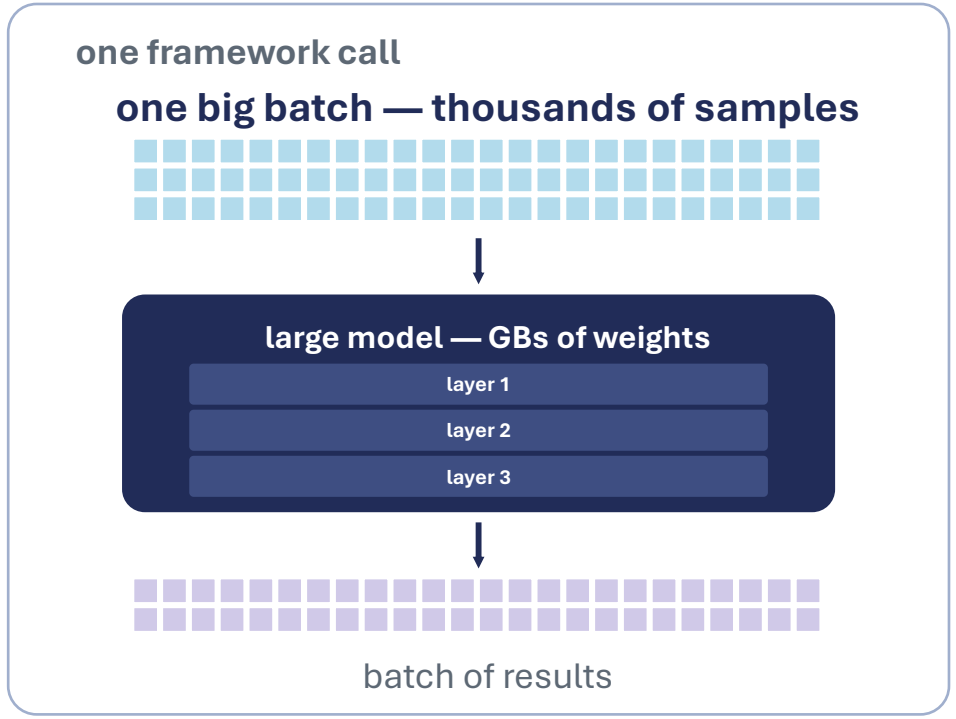


The workload = one tiny Mat-Vec chain × millions of GPU threads, every frame.

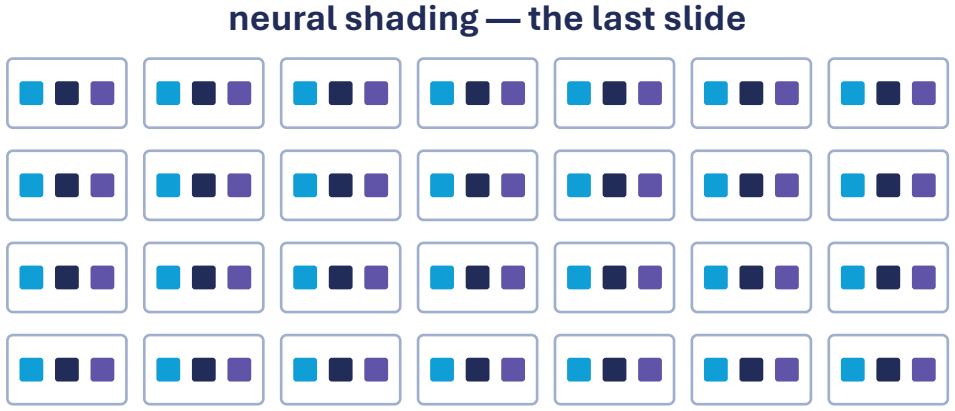
HPG 2026

What an ML framework is built for

In a typical ML problem, the model is the main program.



vs



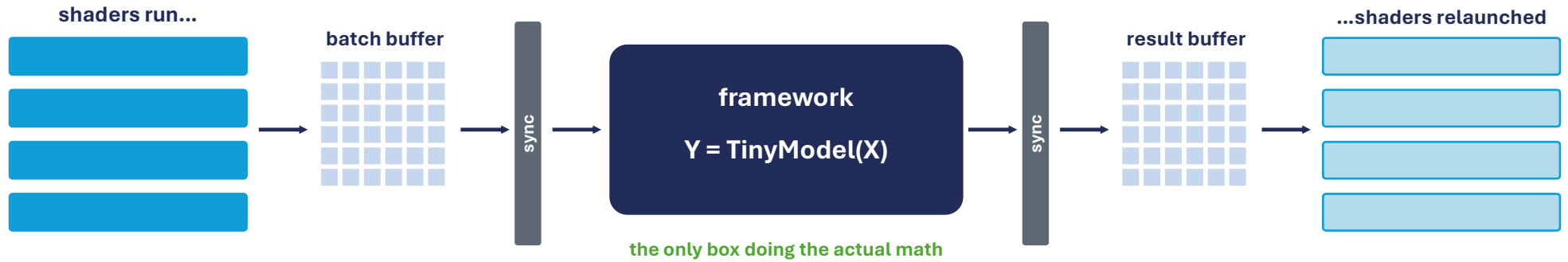
each tiny card = one invocation:
locals → Wx+b chain → result

a huge kernel call, driven by the framework — millions of parallel tiny calls, inside the renderer

HPG 2026

Forcing it into the framework's shape

To use a framework mid-frame, the workload must be reshaped — twice.



the cost, every frame:

shader split in two	locals: registers → DRAM	flush + GPU handover	results: DRAM → registers	shaders relaunched

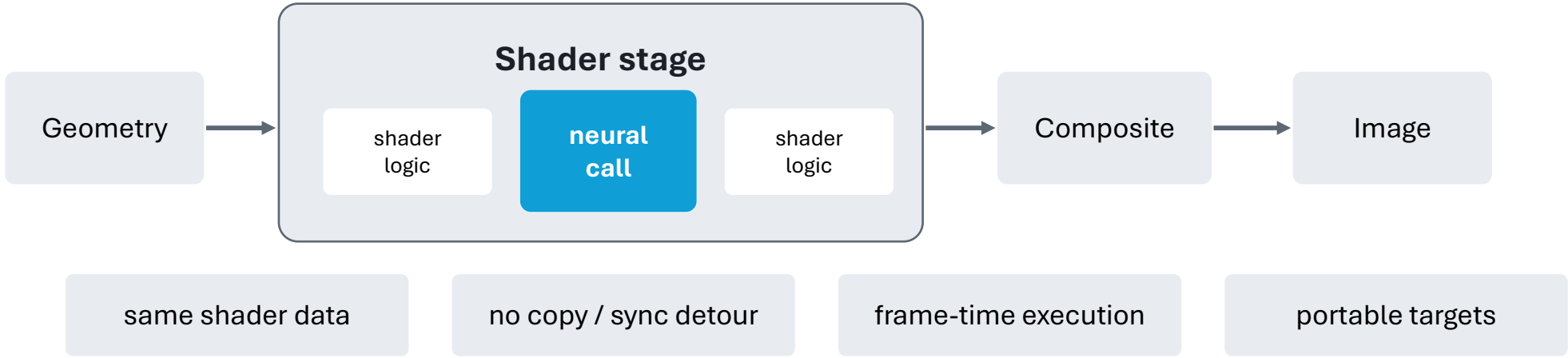
...repeated every frame, for every neural call site in the renderer

All of this plumbing — to deliver a few microseconds of Wx + b.

HPG 2026

The right model: inline neural shaders

Neural code should be callable as a normal shader operation.



Goal: use neural networks without leaving the shader world.

HPG 2026

What does a neural shader need?

If neural code is a normal shader operation, it still needs a few reusable primitives.

Feature encoding

map raw shader inputs
into a neural feature space

Neural evaluation

compute learned functions
inline with shader logic

Differentiability

define once; support training
or fine tuning when needed

Efficient execution

meet real-time budgets
across renderer targets

These are the building blocks we package as shader-language abstractions.

HPG 2026

If we write the neural shader inline by hand

All four needs become code the shader author has to maintain.

forward

```
Color evalTexture(Pos uv)
{
  Feat f = Feat(0.0);

  for (int l = 0; l < 8; ++l) {
    float2 p = uv * scale(l) + offset(l);
    int2 base = int2(floor(p));
    float2 t = frac(p);
    // ... bounds, masks, address math ...

    for (int cy = 0; cy < 2; ++cy)
    for (int cx = 0; cx < 2; ++cx) {
      uint s = hash(base + int2(cx,cy), l);
      float w = bilerp(float2(cx,cy), t);
      f[2*l+0] += w * enc[s+0];
      f[2*l+1] += w * enc[s+1];
    }
  }

  // ... vector packing and storage swizzle ...

  Hidden h;
  for (int o = 0; o < 128; ++o) {
    float a = bias(L0, o);
    for (int i = 0; i < 16; ++i)
      a += weight(L0, o, i) * f[i];
    h[o] = relu(a);
  }

  // ... activation cache and target paths ...

  Hidden h2;
  for (int o = 0; o < 128; ++o) {
    float a = bias(L1, o);
    for (int i = 0; i < 128; ++i)
      a += weight(L1, o, i) * h[i];
    h2[o] = relu(a);
  }

  // ... repeat for variants / layer shapes ...

  Color y;
  for (int o = 0; o < 3; ++o) {
    float a = bias(L2, o);
    for (int i = 0; i < 128; ++i)
      a += weight(L2, o, i) * h2[i];
    y[o] = sigmoid(a);
  }
  // ... output transform and format conversion ...
  return y;
}
```

backward

```
void evalTexture_bwd(Color dY)
{
  // replay or cache forward intermediates
  Feat f; Hidden h, h2;
  Hidden dh = Hidden(0.0);
  Hidden dh2 = Hidden(0.0);
  Feat df = Feat(0.0);
  ...

  for (int o = 0; o < 3; ++o) {
    float dz = dY[o] * sigmoidGrad(y[o]);
    atomicAdd(dBias(L2,o), dz);
    for (int i = 0; i < 128; ++i) {
      atomicAdd(dWeight(L2,o,i), dz*h2[i]);
      dh2[i] += weight(L2,o,i) * dz;
    }
  }

  // ... reductions for dMlp and dh2 ...

  for (int o = 0; o < 128; ++o) {
    float dz = dh2[o] * reluGrad(h2[o]);
    atomicAdd(dBias(L1,o), dz);
    for (int i = 0; i < 128; ++i) {
      atomicAdd(dWeight(L1,o,i), dz*h[i]);
      dh[i] += weight(L1,o,i) * dz;
    }
  }

  for (int o = 0; o < 128; ++o) {
    float dz = dh[o] * reluGrad(h[o]);
    atomicAdd(dBias(L0,o), dz);
    for (int i = 0; i < 16; ++i) {
      atomicAdd(dWeight(L0,o,i), dz*f[i]);
      df[i] += weight(L0,o,i) * dz;
    }
  }

  // ... vectorized and tiled variants ...

  for (int l = 0; l < 8; ++l)
  for (int cy = 0; cy < 2; ++cy)
  for (int cx = 0; cx < 2; ++cx) {
    uint s = replayHash(uv, l, cx, cy);
    float w = replayWeight(uv, l, cx, cy);
    atomicAdd(dEnc[s+0], w * df[2*l+0]);
    atomicAdd(dEnc[s+1], w * df[2*l+1]);
    // ... dUv terms and hash collisions ...
  }
  // ... target-specific atomics/reductions ...
}
```

excerpts only: the full handwritten path is a few hundred lines of code

HPG 2026

The code is the pain

forward

```
Color evalTexture(Pos uv)
{
  Feat f = Feat(0.0);

  for (int l = 0; l < 8; ++l) {
    float2 p = uv * scale(l) + offset(l);
    int2 base = int2(floor(p));
    float2 t = frac(p);
    // ... bounds, masks, address math ...

    for (int cy = 0; cy < 2; ++cy)
    for (int cx = 0; cx < 2; ++cx) {
      uint s = h
      float w =
      f[2*l+0]
      f[2*l+1]
    }
  }

  // ... vector

  Hidden h;
  for (int o =
    float a = b
    for (int i
      a +=
    h[o] = relu
  }

  // ... activation cache and target paths ...

  Hidden h2;
  for (int o = 0; o < 128; ++o) {
    float a = bias(L1, o);
    for (int i = 0; i < 128; ++i)
      a += weight(L1, o, i) * h[i];
    h2[o] = relu(a);
  }

  // ... repeat for variants / layer shapes ...

  Color y;
  for (int o = 0; o < 3; ++o) {
    float a = bias(L2, o);
    for (int i = 0; i < 128; ++i)
      a += weight(L2, o, i) * h2[i];
    y[o] = sigmoid(a);
  }
  // ... output transform and format conversion ...
  return y;
}
```

backward

```
void evalTexture_bwd(Color dY)
{
  // replay or cache forward intermediates
  Feat f; Hidden h, h2;
  Hidden dh = Hidden(0.0);
  Hidden dh2 = Hidden(0.0);
  Feat df = Feat(0.0);
  ...

  for (int o = 0; o < 3; ++o) {
    float dz = dY[o] * sigmoidGrad(y[o]);
    atomicAdd(dBias(L2,o), dz);
    for (int i = 0; i < 128; ++i) {
      atomicAdd(dWeight(L2,o,i), dz*h2[i]);
      dh2[i] += weight(L2,o,i) * dz;
    }
  }

  for (int o = 0; o < 128; ++o) {
    float dz = dh2[o] * reluGrad(h2[o]);
    atomicAdd(dBias(L1,o), dz);
    for (int i = 0; i < 128; ++i) {
      atomicAdd(dWeight(L1,o,i), dz*h[i]);
      dh[i] += weight(L1,o,i) * dz;
    }
  }

  // ... activation cache and target paths ...

  for (int o = 0; o < 128; ++o) {
    float dz = dh[o] * reluGrad(h[o]);
    atomicAdd(dBias(L0,o), dz);
    for (int i = 0; i < 16; ++i) {
      atomicAdd(dWeight(L0,o,i), dz*f[i]);
      df[i] += weight(L0,o,i) * dz;
    }
  }

  // ... vectorized and tiled variants ...

  for (int l = 0; l < 8; ++l)
  for (int cy = 0; cy < 2; ++cy)
  for (int cx = 0; cx < 2; ++cx) {
    uint s = replayHash(uv, l, cx, cy);
    float w = replayWeight(uv, l, cx, cy);
    atomicAdd(dEnc[s+0], w * df[2*l+0]);
    atomicAdd(dEnc[s+1], w * df[2*l+1]);
    // ... dUv terms and hash collisions ...
  }
  // ... target-specific atomics/reductions ...
}
```

😣 **painful to maintain**

wouldn't it be nice if we could abstract this away?

Keep the neural shader inline; move the repetitive machinery behind reusable interfaces.

HPG 2026

neural.slang abstracts this into interfaces

Reusable Slang abstractions, with explicit interfaces for each role.

neural shader need	neural.slang abstraction	interface
Feature encoding	encoder interfaces and implementations	**IEncoder**
Neural evaluation	layer and network APIs	**ILayer**
Differentiability	Slang automatic differentiation	**[Differentiable]**
Efficient execution	portable and accelerated execution	**IStorage & IVector**

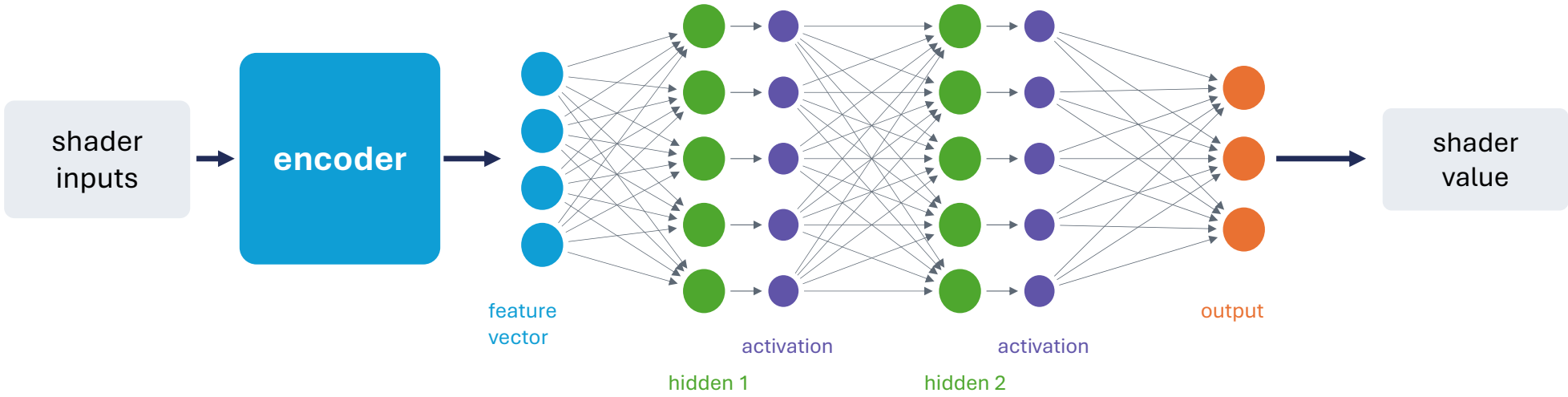
Same shader language; same compilation pipeline; same renderer data flow.

Slang targets CUDA, Vulkan/SPIR-V, Metal, Direct3D, WebGPU, and CPU backends.

HPG 2026

How the interfaces work together

One neural shader function, with interfaces attached to the dataflow.

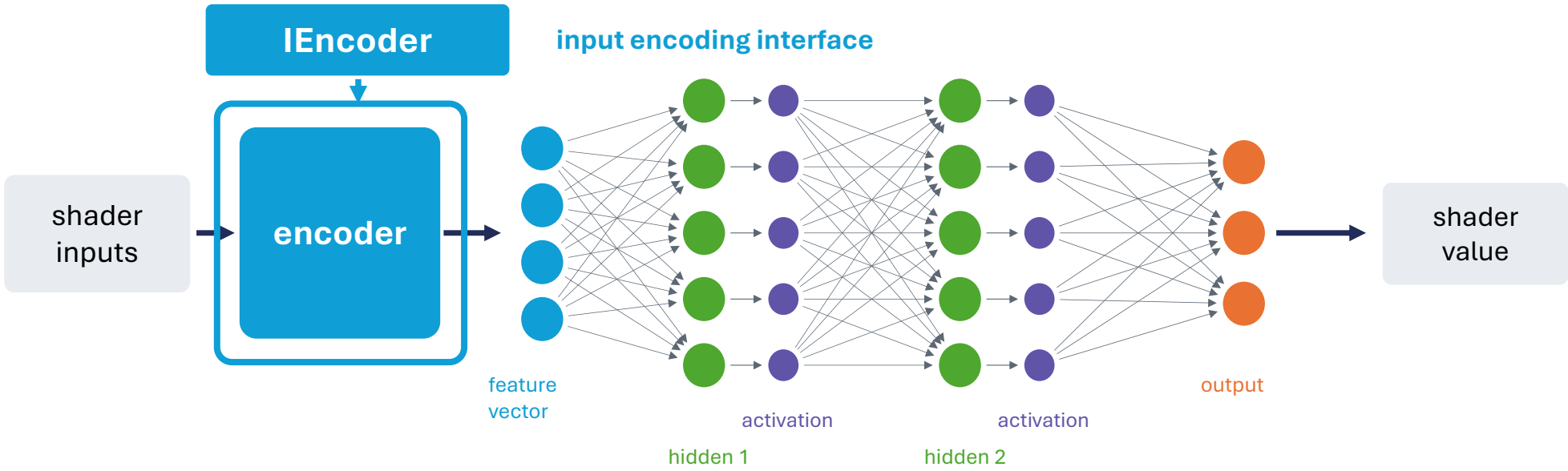


Click through to reveal the interface attached to each part of the network.

HPG 2026

How the interfaces work together

One neural shader function, with interfaces attached to the dataflow.

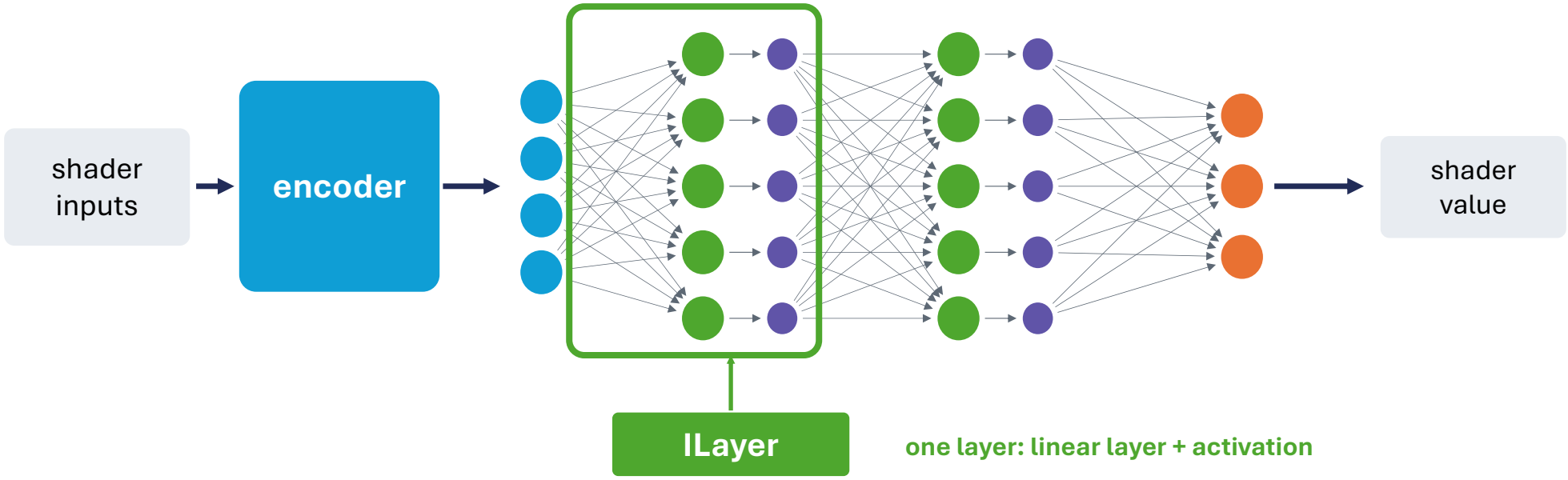


Same network structure; one interface boundary highlighted per build.

HPG 2026

How the interfaces work together

One neural shader function, with interfaces attached to the dataflow.

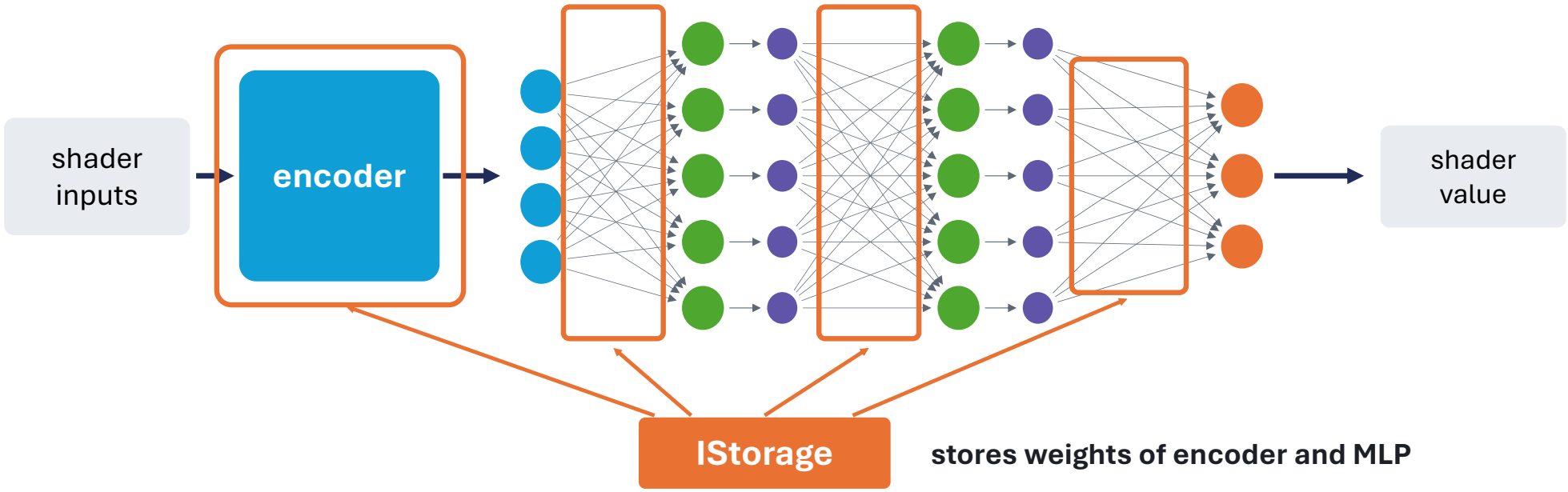


Same network structure; one interface boundary highlighted per build.

HPG 2026

How the interfaces work together

One neural shader function, with interfaces attached to the dataflow.

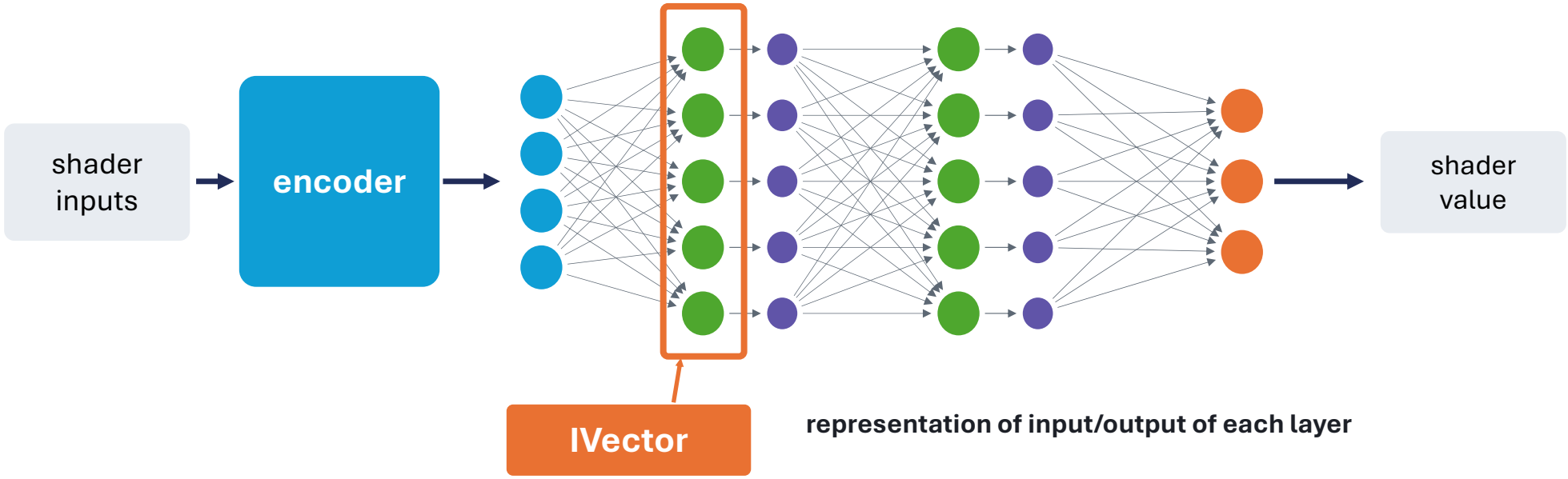


Same network structure; one interface boundary highlighted per build.

HPG 2026

How the interfaces work together

One neural shader function, with interfaces attached to the dataflow.

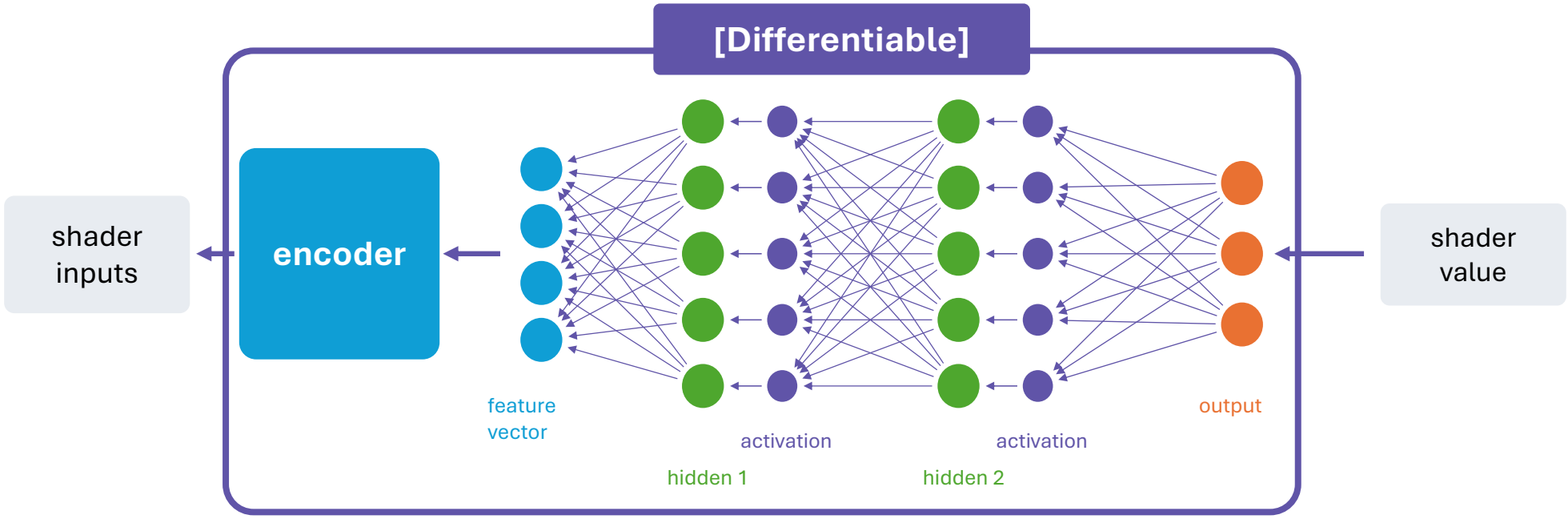


Same network structure; one interface boundary highlighted per build.

HPG 2026

How the interfaces work together

One neural shader function, with interfaces attached to the dataflow.



Same network structure; one interface boundary highlighted per build.

HPG 2026

API structure

Key concrete types implemented in the neural.slang module.

role	interface boundary	current implementations
input encoding	IEncoder	PermutoEncoder
layers + activation	ILayer / IActivation	FFLayer + activations ReLU, Sigmoid, SiLU, QuickGELU, ...
vector execution	IVector	InlineVector: all targets* WaveTangledVector: Vulkan / CUDA / Metal
parameters + layout	IStorage	BindlessAddress, PointerAddress, ... LinearLayout, OptimalLayout

*except WebGPU — parameter storage is bindless-only

Users can add encoders, layers, activations, and vector representations; storage is low-level infrastructure only provided by neural.slang.

HPG 2026

Code example: neural.slang

17 lines define the neural function directly in Slang.

```
import slang.neural;
typealias Pos = InlineVector<float, 2>;
typealias Feat = InlineVector<float, 16>;
typealias Hidden = InlineVector<float, 128>;
typealias Color = InlineVector<float, 3>;
typealias Encoder = PermutoEncoder<float, 2, 2, 16, 8, Pos, Feat>;
typealias InLayer = FFLayer<float, Feat, Hidden, OptimalLayout, ReLU<float>, true>;
typealias HiddenLayer = FFLayer<float, Hidden, Hidden, OptimalLayout, ReLU<float>, true>;
typealias OutLayer = FFLayer<float, Hidden, Color, OptimalLayout, Sigmoid<float>, true>;

[Differentiable]
Color evalTexture(Pos uv, BindlessAddress<float> enc, BindlessAddress<float> mlp)
{
    var f = Encoder(buildHyperParameters()).encode<BindlessAddress<float>>(uv, enc);
    var h = InLayer().eval(f, mlp);
    h = HiddenLayer().eval(h, mlp.getOffset(InLayer.StorageSize));
    return OutLayer().eval(h, mlp.getOffset(InLayer.StorageSize + HiddenLayer.StorageSize));
}
```

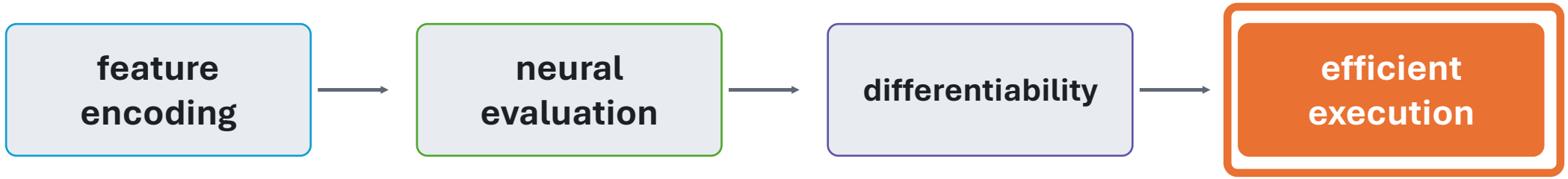
17 nonblank lines

2D coordinate -> encoder -> 16 -> 128 -> 128 -> 3

HPG 2026

The missing piece: efficient execution

The API reduces authoring cost. The next question is whether the generated path can still run at real-time speed.

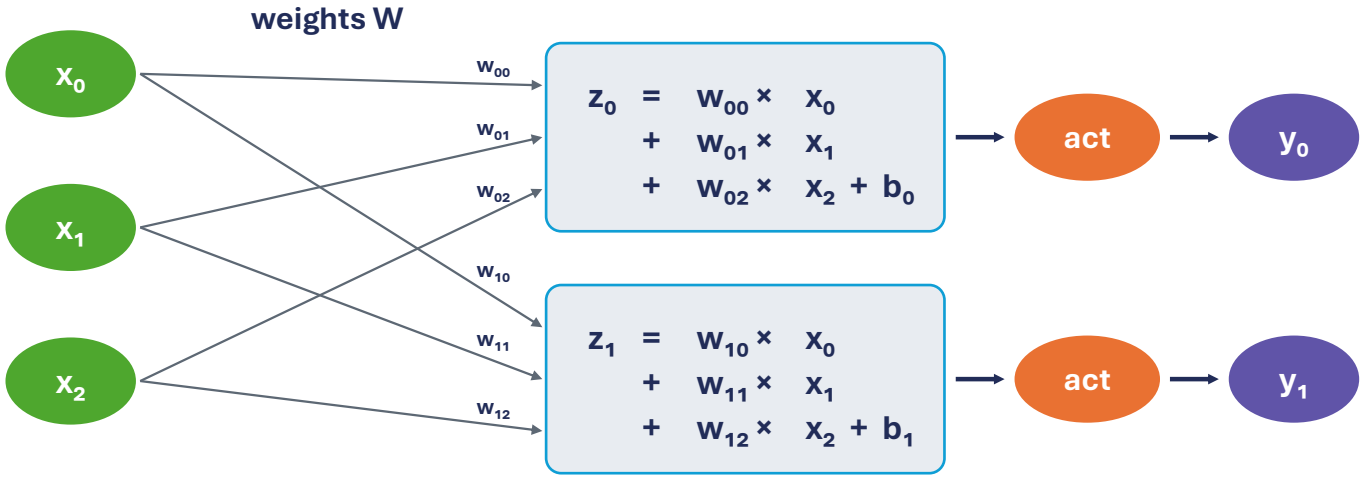


The goal is convenient code that still fits the frame budget.

HPG 2026

Zoom in: one layer

A dense layer forms weighted sums, adds bias, then applies an activation.



$$z = W x + b$$

$$y = \text{activation}(z)$$

Therefore: $y = \text{activation}(Wx + b)$

HPG 2026

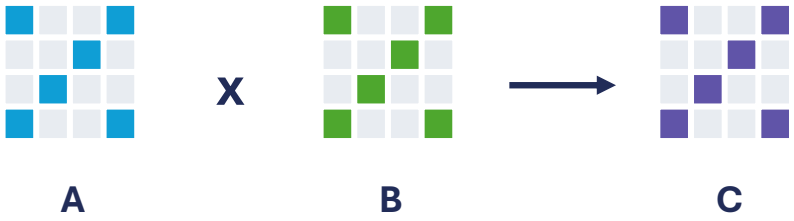
Modern GPUs expose matrix-matrix MMA

NVIDIA Tensor Cores

AMD Matrix Cores

Metal SIMD Matrix

Other matrix accelerators



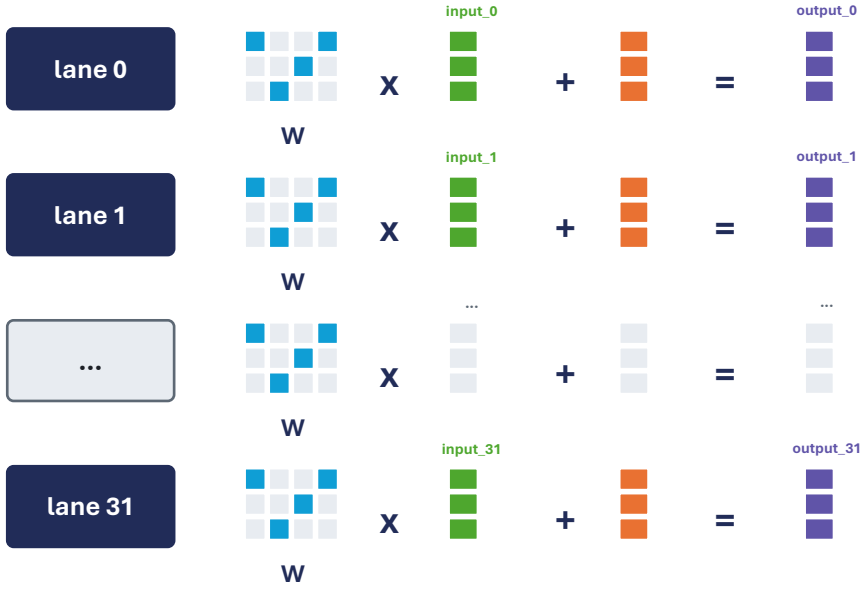
Hardware wants tiled matrix x matrix

But each shader lane naturally has one vector for one layer evaluation.

HPG 2026

Matvec to MMA: per-lane view

SIMT view: each lane evaluates the same layer with its own input vector.



same weights W for all lanes

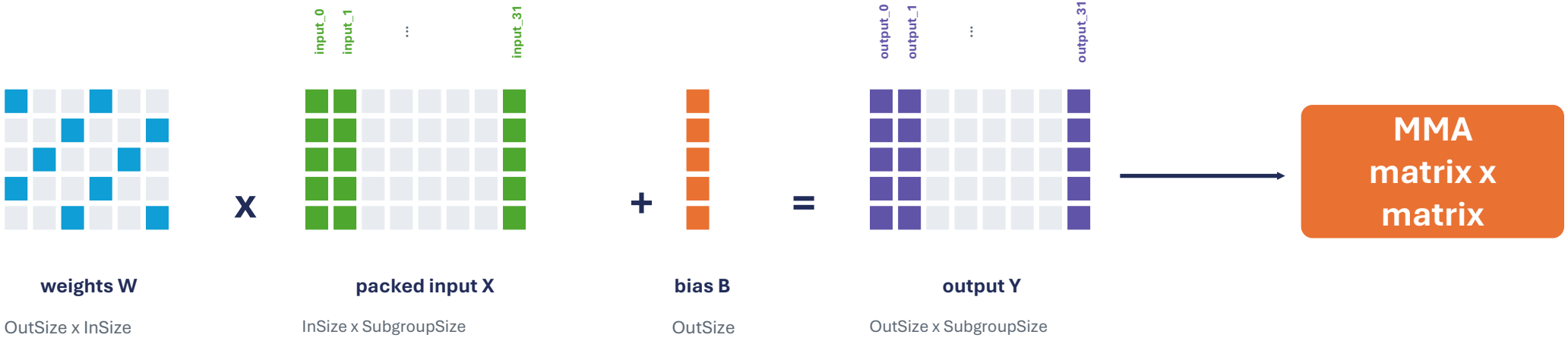
different input vector per lane

This is a batch of independent matrix-vector multiplies from the shader author's point of view.

HPG 2026

Matvec to MMA: packed subgroup view

Cooperative execution packs lane-local vectors into matrix columns.



The hard part is constructing this packed representation without exposing it to the shader author.

HPG 2026

Cooperative Vector hides the packing details

Vulkan/D3D path hides subgroup packing, layout, and MMA scheduling.

Cooperative Vector extension

- D3D - linalg-matrix
- Vulkan – VK_NV_cooperative_vector

shader-facing matrix-vector API

subgroup packing | **matrix layout** | **MMA scheduling**

packing, layout, and scheduling hidden by the API

→

Matrix Hardware

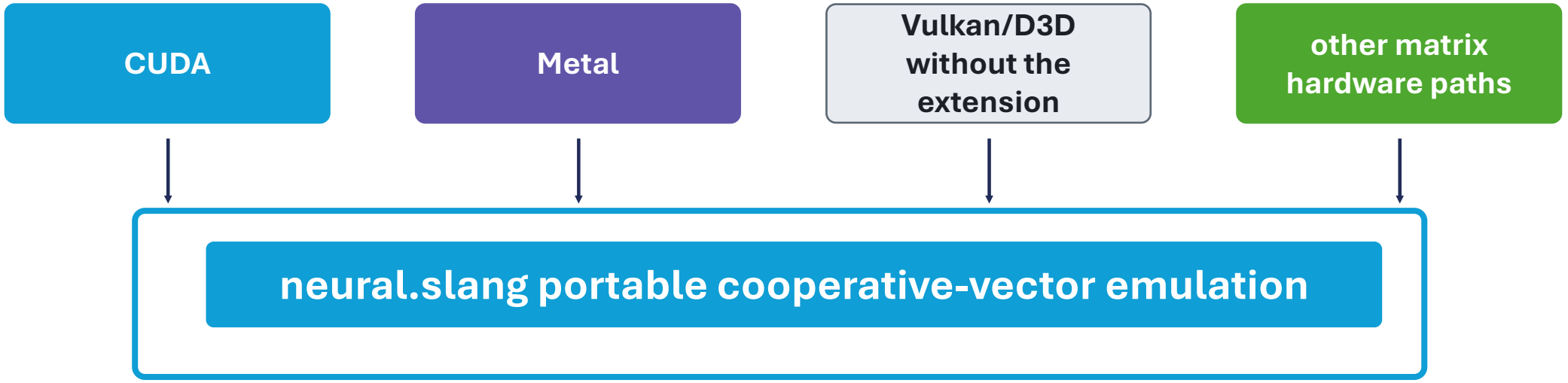
MMA

Great when this extension path exists.

HPG 2026

What if Cooperative Vector is unavailable?

The same shader layer still needs packed-subgroup matvec.

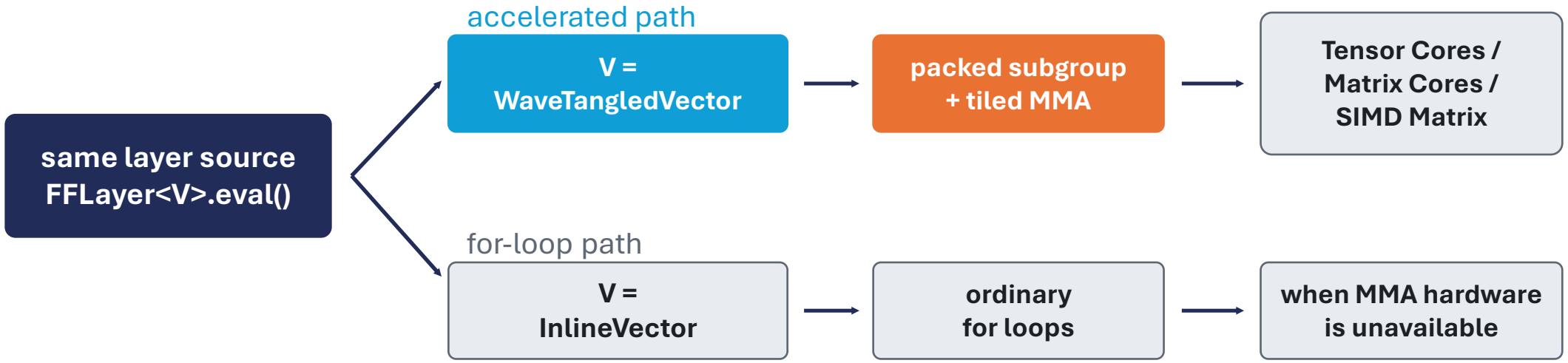


Same layer source; backend-specific code handles packing, tiling, and matrix instructions.

HPG 2026

Vector type selects the execution path

In FFLayer<V>, V is the vector type that supplies matvec.

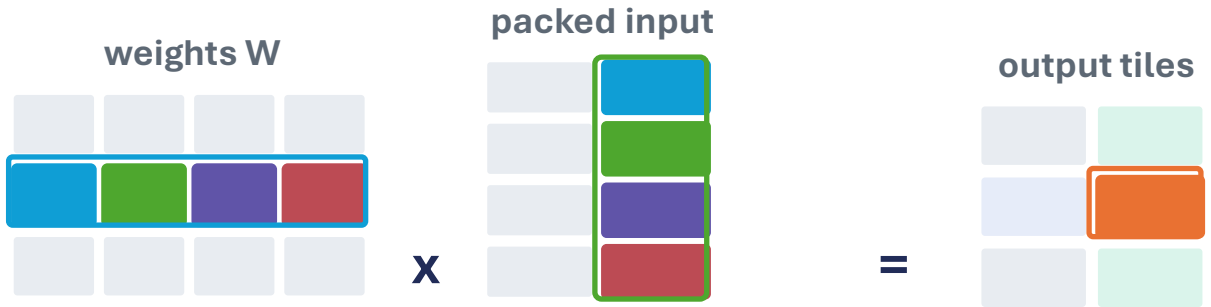


The API stays fixed; the chosen vector type controls how the layer matvec is evaluated.

HPG 2026

WaveTangledVector key point: matrix tiling

Packed subgroup matvec becomes tiled matrix multiply.



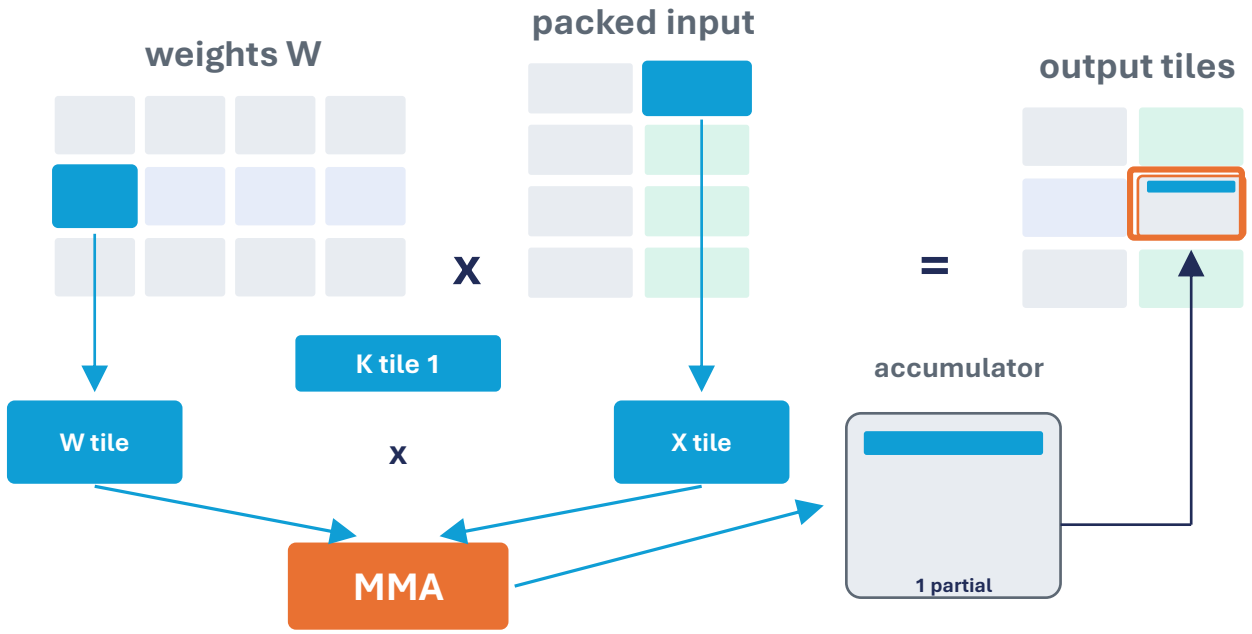
One output tile is built by walking across matching input tiles.

Each click advances one K tile.

HPG 2026

Tiled MMA flow: first tile pair

Load one W tile and one packed-input tile.

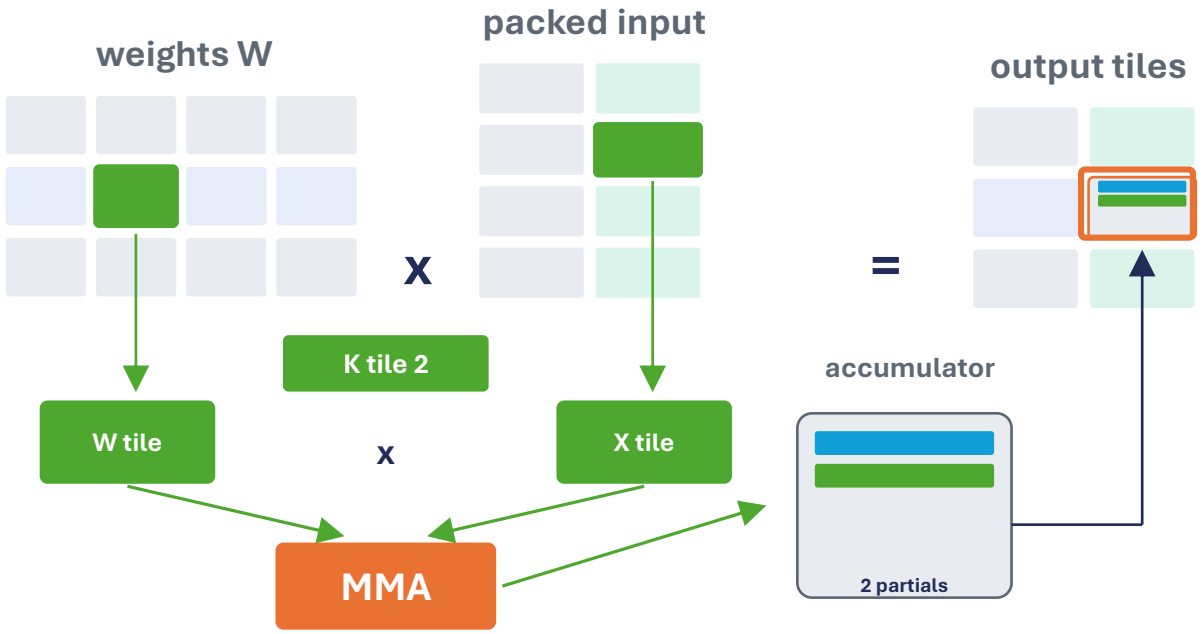


Each step accumulates partials into one output tile.

HPG 2026

Tiled MMA flow: next tile pair

Accumulate another partial product into the same output tile.

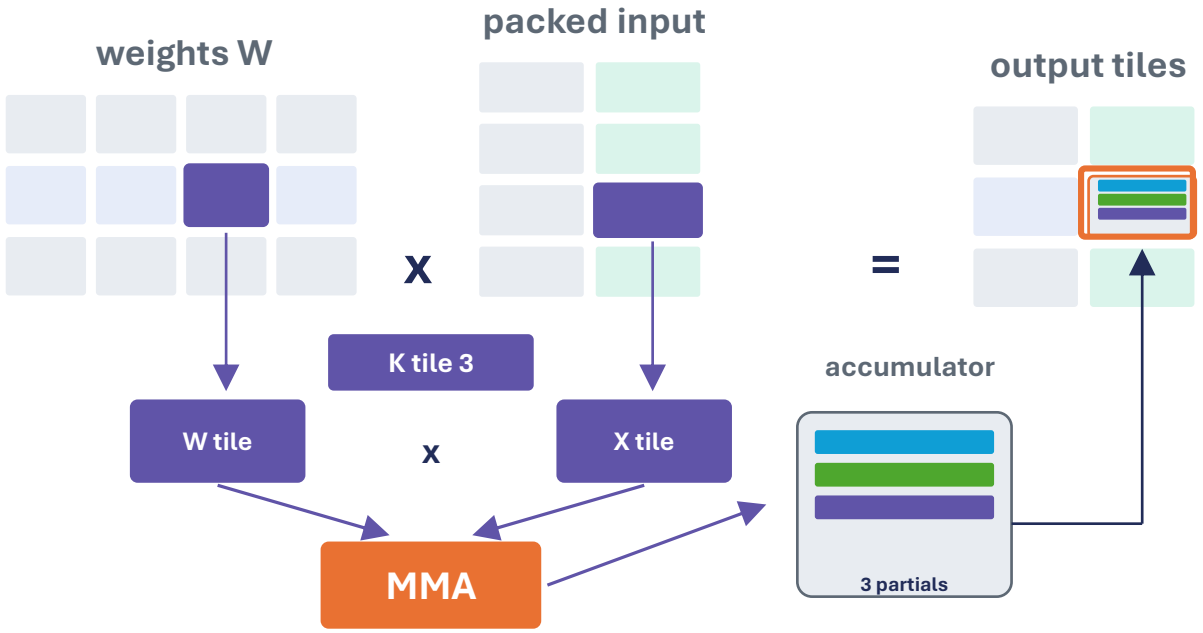


Each step accumulates partials into one output tile.

HPG 2026

Tiled MMA flow: third tile pair

Continue accumulating partial products.

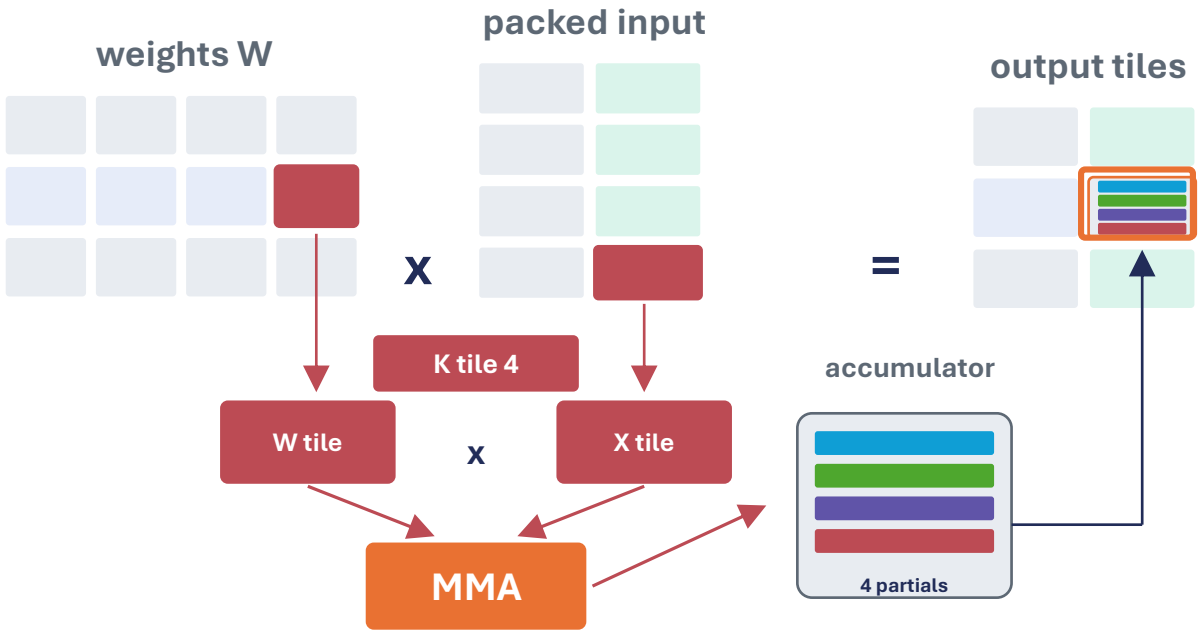


Each step accumulates partials into one output tile.

HPG 2026

Tiled MMA flow: accumulated output tile

After the K loop, the accumulator becomes the output tile.



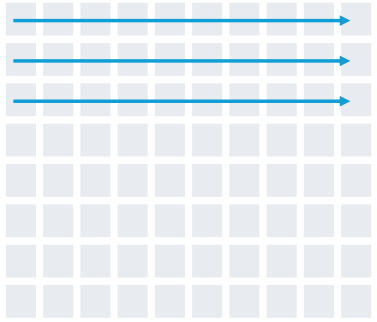
Each step accumulates partials into one output tile.

HPG 2026

WaveTangledVector key point: target-specific layout

Example: row-major -> tile-contiguous -> CUDA thread/register layout

1 row-major matrix



2 tiled MMA layout

3 one CUDA MMA tile

zoom: values are assigned to CUDA threads/registers

Row\Col	0-1	2-3	4-5	6-7
0	thread 0 reg a0,a1	thread 1 reg a0,a1	thread 2 reg a0,a1	thread 3 reg a0,a1
1	thread 4 reg a0,a1	thread 5 reg a0,a1	thread 6 reg a0,a1	thread 7 reg a0,a1
...				
7	thread 28 reg a0,a1	thread 29 reg a0,a1	thread 30 reg a0,a1	thread 31 reg a0,a1
8	thread 0 reg a2,a3	thread 1 reg a2,a3	thread 2 reg a2,a3	thread 3 reg a2,a3
9	thread 4 reg a2,a3	thread 5 reg a2,a3	thread 6 reg a2,a3	thread 7 reg a2,a3
...				
15	thread 28 reg a2,a3	thread 29 reg a2,a3	thread 30 reg a2,a3	thread 31 reg a2,a3

CUDA reference: PTX mma.m16n8k8 per-thread register layout

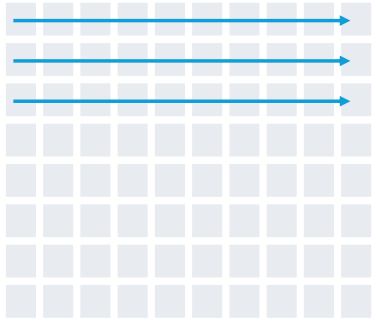
Different targets expect different layouts for their best performance.

HPG 2026

WaveTangledVector key point: target-specific layout

Example: row-major -> tile-contiguous -> CUDA thread/register layout

1 row-major matrix



memory order

2 tiled MMA layout

memory order

3 one CUDA MMA tile

zoom: values are assigned to CUDA threads/registers

Row\Col	0-1	2-3	4-5	6-7
0	thread 0 reg a0,a1	thread 1 reg a0,a1	thread 2 reg a0,a1	thread 3 reg a0,a1
1	thread 4 reg a0,a1	thread 5 reg a0,a1	thread 6 reg a0,a1	thread 7 reg a0,a1
...				
7	thread 28 reg a0,a1	thread 29 reg a0,a1	thread 30 reg a0,a1	thread 31 reg a0,a1
8	thread 0 reg a2,a3	thread 1 reg a2,a3	thread 2 reg a2,a3	thread 3 reg a2,a3
9	thread 4 reg a2,a3	thread 5 reg a2,a3	thread 6 reg a2,a3	thread 7 reg a2,a3
...				
15	thread 28 reg a2,a3	thread 29 reg a2,a3	thread 30 reg a2,a3	thread 31 reg a2,a3

CUDA reference: PTX mma.m16n8k8 per-thread register layout

```
NetworkParameterLayoutConverter<half, 0b111ull, 16, 128, 128, 3>.toOptimalLayout(portable, optimal, tid, threads);
```

HPG 2026

WaveTangledVector key constraint: divergence

WaveTangledVector requires uniform control flow.

- Divergent control flow leaves some lanes inactive — and restoring them is driver-level control that Slang does not have.
- For divergent code, use **Cooperative Vector** where the platform supports it: it handles divergence below the API.
- **Cooperative Vector** backend support in **neural.slang** is on the roadmap — coming soon.

HPG 2026

Training adds more matrix work

For $y = W x + B$, the chain rule gives a transposed matvec, an outer product, and a reduction.

$dx = W^T \times dy$

$dW = dy \times x^T$

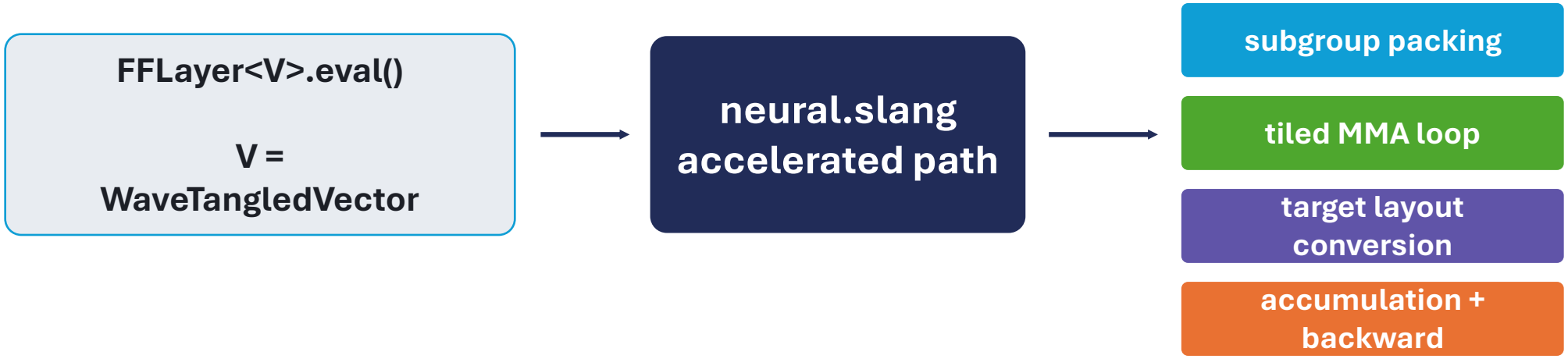
$dB = \text{reduce}(dy)$

Full backward is also an MMA problem.

HPG 2026

The API chooses the performance path

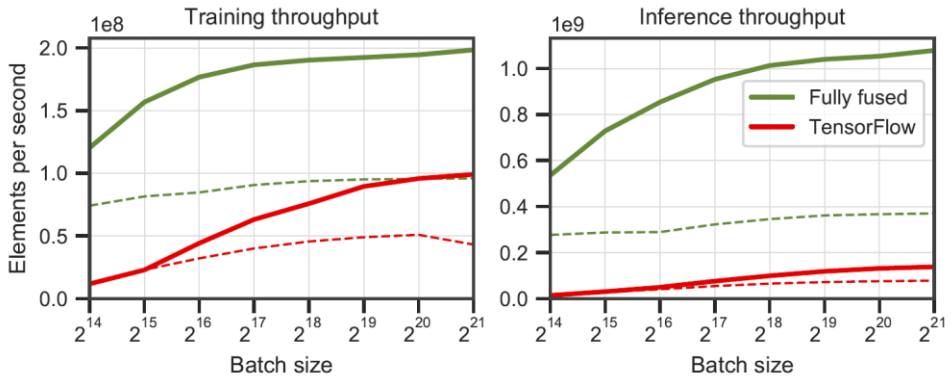
Developers still write one layer call.



The vector type selects the execution strategy; the layer API stays the same.

HPG 2026

Performance baseline

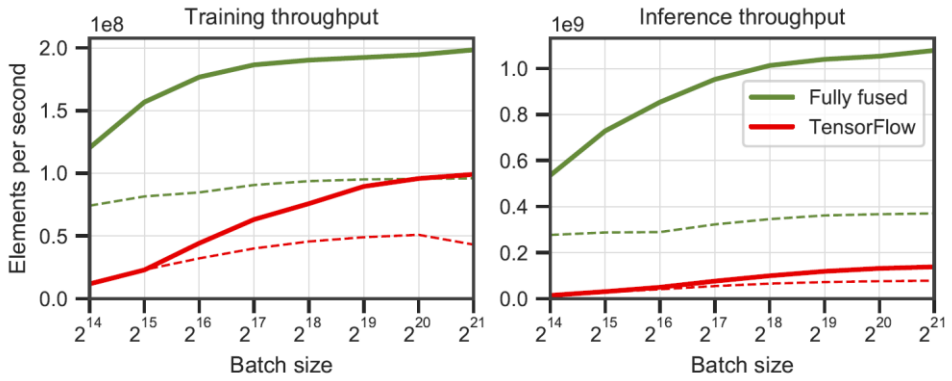


Source: Müller et al., Real-time Neural Radiance Caching for Path Tracing, SIGGRAPH 2021.

Fully fused MLPs are already much faster than general ML frameworks.

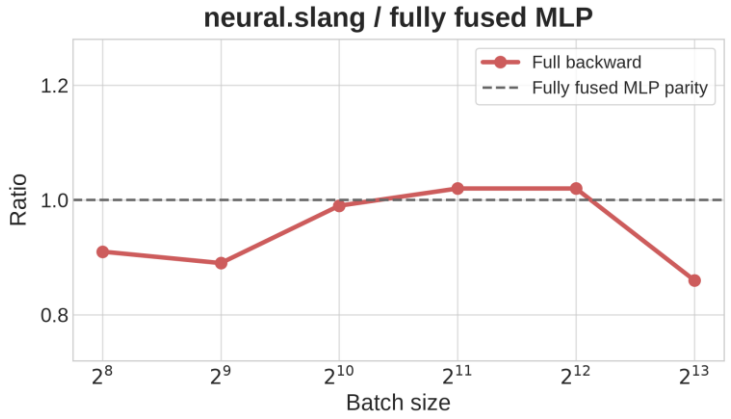
HPG 2026

Performance baseline



Source: Müller et al., Real-time Neural Radiance Caching for Path Tracing, SIGGRAPH 2021.

Fully fused MLPs are already much faster than general ML frameworks.

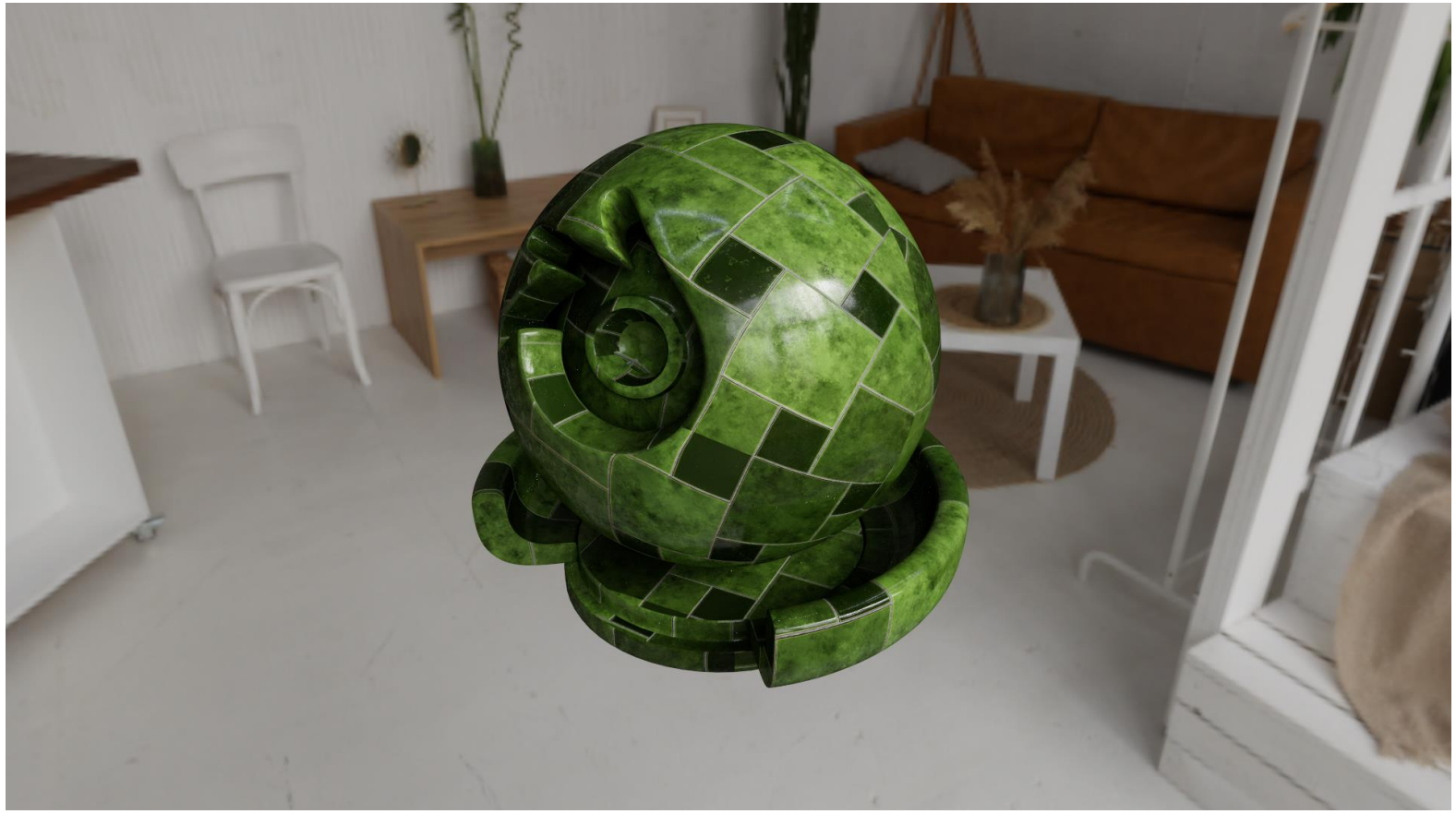


Ratio near 1.0 means fully fused MLP parity.

Shader-native API
same performance class

HPG 2026

Example: neural material in a path tracer



A path tracer with a neural material — and the result is beautiful.

HPG 2026

What developers get

- shader-native MLP authoring
- 17 lines vs 100+
- portable across targets
- for-loop + accelerated paths

neural.slang makes small neural networks practical as shader building blocks.

HPG 2026

More resources

- Siggraph 2026 course:
  An Introduction to Neural Shading
  - 9:00 am – 12:15 am, July 19
  - Room 515A

- Sample code for neural.slang:

  shader-slang/slangpy-samples/experiments/neural_slang

HPG 2026