GDC MARCH 17-21, 2025
SAN FRANCISCO, CA

Cooperative Vectors and Neural Rendering

Shawn Hargreaves – Microsoft
Anis Benyoub – Intel
Joe Rozek – AMD
Alexey Panteleev – NVIDIA

Shader Programming Today

During development

- Identify some interesting computation
- Implement it in HLSL

Runtime

- Implicitly data parallel programming model
- Wide vector registers
- Primarily float32 data types
- Control flow, divergence…

A New Programming Model

During development

- Get training data
- Train a model

= go find many examples of inputs and desired outputs

= automated function approximation

Runtime

- Inference

= matrix convolve

Why is this interesting?

Solve problems where we have example answers but don’t know an analytic solution

Replace expensive computations with cheaper automatically discovered approximations (rendering is full of approximations and coherency)

Generalized function approximators can do more per watt

- Divergence of data (weights) rather than code
- Highly regular memory access patterns
- Highly tolerant of quantization

Where might we apply this?

Expensive shader -> neural material

Large texture -> neural compression

Low resolution / aliased rendering -> neural upscaling or AA

Noisy raytracing -> neural denoiser

xxx -> yyy?

Models don’t have to solve 100% of the problem. Train on the delta between a cheap conventional rendering approximation vs. impractical high-quality solution?

aka: ML fixes up artifacts from other techniques

Introducing: Cooperative Vectors

Cooperative Vectors

New HLSL language feature, part of shader model 6.9

Comes with Direct3D 12 API and PIX debugger support

Takes advantage of specialized vector/matrix acceleration hardware

- Matrix/Vector Multiply
- Matrix/Vector Multiply-Add
- Vector/Vector Outer Product and Accumulate
- Reduce and Accumulate

Matrix Formats

```
enum D3D12_LINEAR_ALGEBRA_DATATYPE
{
    FLOAT16,
    FLOAT32,
    UINT8,
    UINT16,
    UINT32,
    SINT8,
    SINT16,
    SINT32,
    SINT8_PACKED,
    UINT8_PACKED,
    FLOAT_E4M3,     // FP8: 1 sign bit, 4 exp bits, 3 mantissa bits
    FLOAT_E5M2      // FP8: 1 sign bit, 5 exp bits, 2 mantissa bits
}
```

CheckFeatureSupport()

Multiply and Multiply-Add require FLOAT16, SINT8, E4M3, and E5M2

Accumulate only requires FLOAT16

```
enum D3D12_LINEAR_ALGEBRA_MATRIX_LAYOUT
{
    ROW_MAJOR,
    COLUMN_MAJOR,
    INFERENCING_OPTIMAL,
    TRAINING_OPTIMAL
}
```

Opaque layouts are populated via new D3D12 APIs

Uniformity

Works from any shader stage

Works inside non-uniform control flow

- But fastest when everything is uniform and waves fully filled

Shader Example

```
ByteAddressBuffer inputMatrix0;
ByteAddressBuffer inputMatrix1;
ByteAddressBuffer biasVector0;
ByteAddressBuffer biasVector1;

float3 ps_main(Args args) : SV_TARGET
{
  PreProcessing(args);

  const int M = 64;
  const int K = 64;

  // Neural Network computes the output vector
  // using the same input args and trained data
  // in the form of matrices and bias vectors.

  // The input vector is computed from the shader input
  vector<uint32_t, M> inputVector = SomeFunction(args);

  // Below the physical calculations are replaced by
  // NN evaluation. The Matrix and Bias are trained
  // offline and loaded to memory.
  int moffset0 = 32;
  int boffset0 = 64;
  int moffset1 = 128;
  int boffset1 = 256;

  // layer0 = inputVector * inMat0 + biasV0
  // The matrix and bias are loaded from memory at offsets moffset0 and boffset0
  MatrixRef<M, K> inMat0 = {inputMatrix0, moffset0};
  VectorRef<K> biasV0 = {biasVector0, boffset0};
  vector<uint32_t, K> layer0 = MulAdd<uint32_t, K>(inputVector, inMat0, biasV0);
  layer0 = max(layer0, 0u); // Apply activation function

  // layer1 = layer0 * inMat1 + biasV1
  // The matrix and bias are loaded from memory at offsets moffset1 and boffset1
  MatrixRef<M, K> inMat1 = {inputMatrix0, moffset1};
  VectorRef<K> biasV1 = {biasVector0, boffset1};
  vector<uint32_t, K> layer1 = MulAdd<uint32_t, K>(layer0, inMat1, biasV1);
  layer1 = max(layer1, 0u); // Apply activation function

  // output = layer1 * inMat2 + biasV2
  MatrixRef<M, K> inMat2 = {inputMatrix1, 0};
  VectorRef<K> biasV2 = {biasVector1, 0};
  vector<uint32_t, K> output =
    MulAdd<uint32_t, K, Interpretation::UnsignedInt32>(layer1, inMat2, biasV2);

  output = exp(output);

  float3 color;
  color.r = output[0] * args.lightcolor;
  color.g = output[1] * args.lightcolor;
  color.b = output[2] * args.lightcolor;
  return color;
}
```

Anis Benyoub

Intel

(Intel content not published here)

Joe Rozek

AMD

(AMD content not published here)

Alexey Panteleev

NVIDIA

(NVIDIA content not published here)

Next Steps

Spec: aka.ms/cooperative-vectors-spec

Developer preview in late April

Retail release by end of year

Follow our blog: aka.ms/directx

Plug: Thursday @ 9:30, “*DirectX State of the Union*” (Claire Andrews, Adam Miles)

Thank you!

AMD intel® NVIDIA Qualcomm

Questions?