MESH SHADERS IN AMD RDNA™ 3 ARCHITECTURE

MAX OBERBERGER, LOU KRAMER
TRADITIONAL GEOMETRY PIPELINE

- The traditional geometry pipeline was used for over two decades
- Limited flexibility due to fixed function blocks

- The traditional geometry pipeline was used for over two decades
- Limited flexibility due to fixed function blocks
The traditional geometry pipeline was used for over two decades. It has limited flexibility due to fixed function blocks and no access to topology from vertex shader.
TRADITIONAL GEOMETRY PIPELINE

- **Index Buffer**
- **Vertex Buffer**
- **Input Assembler**
- **Hull Shader**
- **Tessellator**
- **Domain Shader**
- **Geometry Shader**
- **Rasterizer**
- **Pixel Shader**

- Buffer: Buffer
- Fixed function: Fixed function
- Required stage: Required stage
- Optional stage: Optional stage

Triangle List, Triangle Fan, Triangle Strip, ...

Float32, Float16, Bytes, ...

Complicated IO

Slow

one vertex in, one vertex out
MESH SHADER PIPELINE

Mesh Shader → Rasterizer → Pixel Shader

- Buffer
- Fixed function
- Required stage
- Optional stage
MESH SHADER PIPELINE

```
void MeshShader(
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES],
)
```

DispatchMesh(4, 3, 2) → Mesh Shader → Rasterizer → Pixel Shader
Mesh Shader Pipeline

DispatchMesh(4, 3, 2) → Mesh Shader → Rasterizer → Pixel Shader

```
void MeshShader(
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES],
    uint threadId : SV_GroupThreadID) {
    verts[threadId] = LoadVertex(threadId);
    tris[threadId] = LoadTriangle(threadId);
}
```

[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]

Buffer | Fixed function | Required stage | Optional stage
Mesh Shader Pipeline

StructuredBuffer<\texttt{uint3}> Triangles : register(t0);
StructuredBuffer<\texttt{float3}> Vertices : register(t1);

\texttt{[NumThreads(128, 1, 1)]}
\texttt{[OutputTopology("triangle")]}?
\texttt{void MeshShader(}
\hspace{1em} \texttt{out indices uint3 tris[MAX_TRIANGLES],}
\hspace{1em} \texttt{out vertices VertexOutput verts[MAX_VERTICES],}
\texttt{)}
MESH SHADER PIPELINE

DispatchMesh(4, 3, 2)

StructuredBuffer<
uint3>

Triangl
es : register(t0);
StructuredBuffer<float3>

Vertices : register(t1);

[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]

void MeshShader(
    out indices uint3 tris[ 256 ],
    out vertices VertexOutput verts[ 256 ],

Buffer

Mesh Shader

Rasterizer

Pixel Shader

Buffer

Fixed function

Required stage

Optional stage
MESH SHADER PIPELINE

DispatchMesh(4, 3, 2) → Mesh Shader → Rasterizer → Pixel Shader

StructuredBuffer<
uint3> Triangles : register(t0);
StructuredBuffer<float3> Vertices : register(t1);
StructuredBuffer<Meshlet> Meshlets : register(t2);

[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]

void MeshShader(
    out indices uint3 tris[ 256 ],
    out vertices VertexOutput verts[ 256 ],

Buffer Generation → Buffer

Mesh Shader Pipeline Diagram

- Buffer
- Fixed function
- Required stage
- Optional stage
So you want to write a mesh shader

```cpp
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {

    // Now what?
}
```
COMPRESSION

```
struct Vertex {
    float3 position;
    float3 normal;
    float2 texCoord;
};

struct Triangle {
    uint32_t v0;
    uint32_t v1;
    uint32_t v2;
};
```

Mesh Size

- Uncompressed Mesh
  - Vertices: 60 MiB
  - Indices: 40 MiB

Buffer Size
COMPRESSION

Mesh Size

Buffer Size

Uncompressed Mesh

Meshlets

Vertices
Indices

40 MiB
40 MiB

60 MiB
73 MiB

1:1
1:1.22
COMPRESSION

18 bit
Global Quantization

~16 bit
Local Quantization

Mesh Size

Buffer Size

Uncompressed Mesh  Meshlets

40 MiB  1:1
60 MiB  1.6:1
37 MiB

Vertices  Indices

Local Quantization

~16 bit
COMPRESSION

2012

Quirin Nikolaus Meyer
Real-Time
Geometry Decompression on
Graphics Hardware
Dissertation

Mesh Size

Buffer Size

Uncompressed Mesh

Meshlets

60 MiB

1:1

40 MiB

1.6:1

37 MiB

Vertices

Indices
**Mesh Size**

- Uncompressed Mesh: 40 MiB
- Meshlets: 60 MiB
- Meshlets (1.6:1): 37 MiB

**Vertex & Triangle counts are limited to 256 elements each**

```cpp
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[256],
    out vertices VertexOutput verts[256],
)
```

**32 Bit Index**

- 96 Bits per Triangle
Vertex & Triangle counts are limited to 256 elements each

```c
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    out indices uint3 tris[256],
    out vertices VertexOutput verts[256],
) {
}```

8 Bit Index
8 Bit Index
8 Bit Index

24 Bits per Triangle
COMPRESSION

Mesh Size

Buffer Size

Uncompressed Mesh

Meshlets

Vertices

Indices

- Uncompressed Mesh
- Meshlets

- 40 MiB
- 60 MiB
- 4:1
- 1.6:1
- 37 MiB
- 10 MiB
8 Bit Index
1 Bit Code
9 Bits per Triangle
COMPRESSION

8 Bit Index
1 Bit Code
9 Bits per Triangle
COMPRESSION

8 Bit Index
1 Bit Code
9 Bits per Triangle
COMPRESSION

- 1 Bit Code
- 1 Bit Code
- ~3 Bit Index
- ~5 Bits per Triangle

Uncompressed Mesh
Buffer Size

Vertices | Indices

Meshlets

Uncompressed Mesh
Meshlets

Mesh Size

Buffer Size

1.6:1
16:1

60 MiB
37 MiB
2.48 MiB
40 MiB
### COMPRESSION

#### Render Time

<table>
<thead>
<tr>
<th>Mesh Type</th>
<th>Render Time in ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Pipeline</td>
<td>0.65</td>
</tr>
<tr>
<td>Meshlets</td>
<td>0.48</td>
</tr>
<tr>
<td>Compressed Meshlets</td>
<td>0.60</td>
</tr>
</tbody>
</table>

#### Mesh Size

<table>
<thead>
<tr>
<th>Mesh Type</th>
<th>Buffer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed Mesh</td>
<td>60 MiB</td>
</tr>
<tr>
<td>Meshlets</td>
<td>16:1</td>
</tr>
<tr>
<td>Compressed Meshlets</td>
<td>2.48 MiB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Uncompressed Mesh</th>
<th>Meshlets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed Lucy</td>
<td>40 MiB</td>
</tr>
<tr>
<td>Meshlets</td>
<td>16:1</td>
</tr>
<tr>
<td>Compressed Lucy</td>
<td>2.48 MiB</td>
</tr>
</tbody>
</table>

- **Rock**: 15.5 M Triangles, 7.8 M Vertices, 75 k Meshlets
- **Lucy**: 28 M Triangles, 14 M Vertices, 136 k Meshlets

GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1; Meshes: Layered Rock by Aixterior, Lucy by Stanford Computer Graphics Laboratory

See endnotes for full test system configuration.
COMPRESSION

Keep an eye on GPUOpen.com

Follow us on X
https://twitter.com/GPUOpen

Follow us on Mastodon
https://mastodon.gamedev.place/@gpuopen

Follow us on Zhihu
https://www.zhihu.com/org/gpuopen-7
AMPLIFICATION SHADERS

DispatchMesh(4, 3, 2)
AMPLIFICATION SHADERS

void AmplificationShader(uint dtid : SV_DispatchThreadID) {
...
    DispatchMesh(x, y, z, payload);
}

DispatchMesh(4, 3, 2)

AMPLIFICATION
SHADER

Mesh Shader

Rasterizer

Pixel Shader
AMPLIFICATION SHADERS

```c
[NumThreads(64, 1, 1)]
void AmplificationShader(uint dtid : SV_DispatchThreadID) {
  ...
  DispatchMesh(x, y, z, payload);
}
```

DispatchMesh(4, 3, 2)
AMPLIFICATION SHADERS – DYNAMIC LOD

```c
[NumThreads(64, 1, 1)]
void AmplificationShader(uint dtid : SV_DispatchThreadID,
                        uint gtid : SV_GroupThreadID)
{
    uint lod, meshletCount;
    ComputeLevelOfDetail(instanceInfo[dtid], lod, meshletCount);
    payload.lod[gtid] = lod;
    DispatchMesh(WaveActiveSum(meshletCount), 1, 1, payload);
}
```

DispatchMesh(4, 3, 2)
AMPLIFICATION SHADERS – CULLING

```
[NumThreads(64, 1, 1)]
void AmplificationShader(uint dtid : SV_DispatchThreadID)
{
    bool visible = IsMeshletVisible(Meshlets[dtid]);
    if (visible) {
        const uint index = WavePrefixCountBits(visible);
        payload.meshletIds[index] = dtid;
    }
    DispatchMesh(WaveActiveCountBits(visible), 1, 1, payload);
}
```

DispatchMesh(4, 3, 2)
**COMPRESSION**

**Render Time**

<table>
<thead>
<tr>
<th></th>
<th>Rock</th>
<th>Lucy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>0.65</td>
<td>1.16</td>
</tr>
<tr>
<td>Meshlets</td>
<td>0.48</td>
<td>0.84</td>
</tr>
<tr>
<td>Compressed Meshlets</td>
<td>0.6</td>
<td>1.07</td>
</tr>
</tbody>
</table>

**Mesh Size**

<table>
<thead>
<tr>
<th></th>
<th>Uncompressed Mesh</th>
<th>Meshlets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Size</td>
<td>40 MiB</td>
<td>16:1</td>
</tr>
<tr>
<td>Mesh Size</td>
<td>60 MiB</td>
<td>1.6:1</td>
</tr>
<tr>
<td>Vertices</td>
<td>2.48 MiB</td>
<td>37 MiB</td>
</tr>
<tr>
<td>Indices</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Rock**
  - 15.5 M Triangles
  - 7.8 M Vertices
  - 75 k Meshlets

- **Lucy**
  - 28 M Triangles
  - 14 M Vertices
  - 136 k Meshlets

GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1; Meshes: Layered Rock by Aixterior, Lucy by Stanford Computer Graphics Laboratory

See endnotes for full test system configuration.
COMPRESSION

Render Time with Culling

- Rock: 15.5 M Triangles, 7.8 M Vertices, 75 k Meshlets
  - Render Time: 0.65 ms
- Lucy: 28 M Triangles, 14 M Vertices, 136 k Meshlets
  - Render Time: 1.16 ms

Mesh Size

- Uncompressed Mesh: 40 MiB
  - Buffer Size: 100 MiB
- Meshlets: 60 MiB
  - Buffer Size: 40 MiB
- Compressed Meshlets: 2.48 MiB
  - Buffer Size: 16 MiB

GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1; Meshes: Layered Rock by Aixterior, Lucy by Stanford Computer Graphics Laboratory

See endnotes for full test system configuration.
QUADRILATERAL PRIMITIVE RASTERIZATION

Meshlet Generation → Buffer → Mesh Shader → Rasterizer → Pixel Shader
QUADRILATERAL PRIMITIVE RASTERIZATION

3D artists use quads for modelling

Meshlet Generation -> Buffer

Mesh Shader -> Rasterizer -> Pixel Shader
QUADRILATERAL PRIMITIVE RASTERIZATION

3D artists use quads for modelling

Meshlet Generation → Buffer → Mesh Shader → Rasterizer → Pixel Shader
QUADRILATERAL PRIMITIVE RASTERIZATION

Barycentric Interpolation

Bilinear Interpolation
QUADRILATERAL PRIMITIVE RASTERIZATION

Bilinear Interpolation
QUADRILATERAL PRIMITIVE RASTERIZATION

Solution: Primitive Attributes!

```c
void MeshShader(
  out indices uint3 tris[MAX_TRIANGLES],
  out primitives PrimitiveAttributes prims[MAX_TRIANGLES],
  out vertices VertexOutput verts[MAX_VERTICES],
)
```
Implementing Bilinear Interpolation using the Mesh Shader approach has certain advantages:

- Flexibility in how mesh data are passed to the graphics pipeline
  - We can create meshlets with quadrilateral primitives underneath
- Some calculations for the bilinear interpolation are done in Mesh Shader and some in Pixel Shader
  - Efficient data exchange between these two stages is important
  - Data can be shared per vertex but also per primitive
QUADRILATERAL PRIMITIVE RASTERIZATION

Bilinear Interpolation

Faster than Tessellation/Geometry pipeline!

Performance of three implementations of bilinear interpolation

GPU: AMD Radeon™ RX 7900 XTX; driver version: 24.1.1

See endnotes for full test system configuration.
PROCEDURAL GEOMETRY

Meshlet Generation → Buffer → Mesh Shader → Rasterizer → Pixel Shader
PROCEDURAL GEOMETRY

```cpp
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {
...
    verts[threadId] = EvaluateGrassSpline(basePosition, threadId);
    tris[threadId] = ComputeTopology(threadId);
}
```
PROCEDURAL GEOMETRY

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {
    ...
    verts[threadId] = EvaluateGrassSpline(basePosition, threadId);
    tris[threadId] = ComputeTopology(threadId);
}
```
PROCEDURAL GEOMETRY

Heightmap

DispatchMesh(x, y, 1)

Amplification Shader

Mesh Shader

Rasterizer

Pixel Shader

void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {
    ...}

verts[threadId] = EvaluateGrassSpline(basePosition, threadId);
tris[threadId] = ComputeTopology(threadId);
50+ million triangles
< 3 milliseconds
PROCEDURAL GEOMETRY

Learn more on GPUOpen.com

Follow us on X
https://twitter.com/GPUOpen

Follow us on Mastodon
https://mastodon.gamedev.place/@gpuopen

Follow us on Zhihu
https://www.zhihu.com/org/gpuopen-7
PERFORMANCE CONSIDERATIONS
PERFORMANCE CONSIDERATIONS - AGENDA

GEOMETRY ENGINE

Command Processor
Geometry Engine
Shader Processor Input

MESH SHADER

AMPLIFICATION (TASK) SHADER
GEOMETRY ENGINE – VERTEX REUSE

- The geometry engine holds a cache for vertex reuse
- Avoids re-shading vertices
- But depends on fixed input structure of vertex pipeline

- Asset pipelines can pack the vertices to make the best use of vertex reuse
- Mesh shaders would not take advantage of this and potentially re-shade vertices multiple times

Mesh shader pipeline might run slower than traditional vertex/pixel shader pipeline!

| VS: 😊 | 0 | 1 | 2 | 2 | 3 | 0 | 1 | 4 | 5 | 5 | 2 | 1 | 6 | 5 | 4 | 4 | 7 | 6 |
|        |  8 shaded, 10 re-used |

| MS: 😁 | 0 | 1 | 2 | 2 | 3 | 0 | 1 | 4 | 5 | 5 | 2 | 1 | 6 | 5 | 4 | 4 | 7 | 6 |
|        | 18 shaded |
GEOMETRY ENGINE – VERTEX REUSE

- Vertex reuse optimization has to be done during meshlet generation

  VS Index Buffer: 0 1 2 2 3 0 1 4 5 5 2 1 6 5 4 4 7 6
  Meshlet Index Buffer: 0 1 2 2 3 0 1 4 5 5 2 1 6 5 4 4 7 6
  Meshlet Unique Vertices Buffer: 0 1 2 3 4 5 6 7

  8 shaded vertices
  6 primitves
  8 vertices

- MS thread reads directly from the buffers
- MS thread group size is max 128 (DirectX® 12)
- MS max output: 256 vertices and 256 primitives (DirectX® 12)

- The fixed function vertex reuse cache is typically only for 32 vertices
Border vertices might need to be duplicated since they fall into multiple meshlets

→ Mesh Shader potentially need to process less vertices in total compared to VS
MESHLET GENERATION – PART 1

- There are different metrics in how meshlets can be constructed
- Depending on the problem case, one or the other is more suitable (content-specific!)

Common metrics that can be considered are

- Number of border vertices
  - Have to be duplicated
- Size of bounding box
  - Helpful for culling
  - Quantization precision
- Triangle strips (topologically connected triangles)
  - Can help with compression rate
  - Might lower performance

- Meshlets can be a collection of loose vertices/primitives
- There’s still lots of room for research in this area
GEOMETRY ENGINE

The GE for Vertex Shaders
- Determines vertexID for each vertex
- Vertex re-use cache
- Prepares the shader export
- Initiate launch of shaders

The GE for Mesh Shaders
- Determines threadID for each thread
- Prepares the shader export
- Initiate launch of shaders

→ Launch rate for mesh shader is faster
GEOMETRY ENGINE

- Prepares the shader export

  - Allocates enough space for the maximum number of exported vertices and primitives per thread group

    ```
    [NumThreads(128, 1, 1)]
    [OutputTopology("triangle")]
    void MeshShader(
        out indices uint3 tris[MAX_TRIANGLES],
        out vertices VertexOutput verts[MAX_VERTICES],
    )
    ```

  - If the export buffer is full, no new waves can be launched
  - Can limit the max. occupancy of mesh shaders

- Space in the shader export is finite
- Designed for “average” mesh shader workloads to reach rasterizer triangle throughput

Allocates enough space for MAX_TRIANGLES primitives and MAX_VERTICES vertices
• An occupancy of ~25% can be enough to reach the triangle throughput limit

• Complex mesh shaders might see higher occupancy (~50%)
• Tend to get limited by available memory in the shader export
• MAX_TRIANGLES and MAX_VERTICES should be set as low as possible

• Mesh shader occupancy can also be limited by
  • VGPRs
  • Group shared memory (LDS)
  • Launch rate
An indicator for a mesh shader being limited by rasterizer throughput is a high export instruction latency, particularly on the first exp instruction. This can be inspected in RGP under the instruction timing tab.
MESH SHADER

• Work a lot like a compute shader

• \(\text{numthreads}(x,y,z)\) to define thread group size

• ThreadID used to read from the input buffer and write to the output buffer

• These are shared buffers

• Any thread can read and write from and to any index

• Unlike a compute shader, the vertices and primitives are exported to the shader export

```cpp
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void MeshShader(
    uint threadId : SV_GroupThreadID,
    out indices uint3 tris[MAX_TRIANGLES],
    out vertices VertexOutput verts[MAX_VERTICES]) {
    ...
    SetMeshOutputCounts(meshlet.vertex_count, meshlet.triangle_count);
    if (threadId < meshlet.vertex_count)
        verts[threadId] = ProcessVertex(vertex_buffer[threadId]);
    if (threadId < meshlet.triangle_count)
        tris[threadId] = ProcessPrimitive(triangle_buffer[threadId])
}
```

• That’s what the geometry engine allocates space for 😊

• Vertex attributes are exported to memory via a regular buffer store (AMD RDNA™3)
  • On AMD RDNA™2 it’s via exp param
**MESH SHADER - EXPORT**

- On AMD RDNA™, the order of primitives and vertices in the shader export is defined by the order of threads in the thread group
- This means thread n exports vertex n and primitive n
  - This is how vertex shaders export their vertices!

```
if (threadId < meshlet.vertex_count)
    verts[threadId] = ProcessVertex(vertex_buffer[threadId]);
if (threadId < meshlet.triangle_count)
    tris[threadId] = ProcessPrimitive(triangle_buffer[threadId])
```

- Unfortunately, this clashes with the flexibility of mesh shaders
  - Any thread can write to any index in the output buffers
MESH SHADER - EXPORT

- Mesh shaders can do something like this:

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

- This does not comply with the order of primitives and vertices in the shader export

- We need to fix this somehow
Mesh shaders can do something like this:

- Group shared memory (LDS) is used to fix this!

- Can be seen in the ISA:

```
ds_load_b32
a_mov_b32
s_mov_t32
s_waitcnt
exp

v0, v0
m0, a3
lgkmcnt(0)
prim v0, off, off, off done row_en
```
MESH SHADER - EXPORT

- A thread can also export multiple vertices and primitives

```
Vertices[thread_id.x] = UnpackVertex(meshlet, thread_id.x);
Vertices[thread_id.x + 8] = UnpackVertex(meshlet, thread_id.x + 8);
```

- On AMD RDNA™3, this is achieved via a wave-wide offset to the export instruction
MESH SHADER - EXPORT

• A thread can also export multiple vertices and primitives

Thread:

```
0 1 2 3 4 5 6 7
```

LDS buffer:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

 Vertex:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

• On AMD RDNA™3, this is achieved via a wave-wide offset to the export instruction
• The compiler may choose to use group shared memory as a staging buffer
MESH SHADER – EXPORT AMD RDNA™2

• A small note for AMD RDNA™2: there is no wave-wide offset
• A thread can only export 1 vertex and 1 primitive max

• Shadow threads are launched that only export
MESH SHADER – EXPORT CONCLUSION

• Thread n should export vertex n and primitive n
• If multiple vertices and primitives are exported per thread, use a wave-wide offset

• Otherwise:
  • Latency increases → can decrease triangle throughput
  • Group shared memory usage increases

• Group shared memory can also be used explicitly in the HLSL shader
  • Efficient data exchange between threads within a thread group is possible

• Group shared memory is limited!
  → Can affect both, mesh shader and pixel shader occupancy
• If not enough group shared memory is available, scratch memory is used
MESHLET GENERATION - PART 2

• What is a recommended configuration?
• There are 2 quite common configurations:

  • V = 128, T = 256
  • Thread group size = 128
    → Larger meshlets

  • V = 64, T = 126 (or 128)
  • Thread group size = 64
    → Smaller meshlets
MESHLET GENERATION - PART 2

• What is a recommended configuration?
• There are 2 quite common configurations:

• \( V = 128, T = 256 \) → Larger meshlets
  More primitives than vertices
  Main compute workload is typically vertex transformations
  The number of primitives should not be the limiting factor, but the number of vertices

• \( V = 64, T = 126 \) (or 128) → Smaller meshlets
  The number of primitives should not be the limiting factor, but the number of vertices

• Thread group size = 128
• Thread group size = 64
MESHLET GENERATION - PART 2

• What is a recommended configuration?
• There are 2 quite common configurations:

  • $V = 128, T = 256$
  • Thread group size = 128 → Larger meshlets
  • $V = 64, T = 126$ (or 128)
  • Thread group size = 64 → Smaller meshlets

  - Thread group size = maximum number of vertices
  - Main compute workload is typically vertex transformations
  - Ensures 1 thread processes 1 vertex
  - If this is not possible, try to divide the maximum number of vertices evenly across the threads
MESHLET GENERATION - PART 2

- What is a recommended configuration?
- There are 2 quite common configurations:

  - V = 128, T = 256
  - Thread group size = 128 → Larger meshlets
  - Larger meshlets have less border vertices in total

  - V = 64, T = 126 (or 128)
  - Thread group size = 64 → Smaller meshlets
  - More border vertices
  - A single thread group needs less resources
  - If occupancy is limited by other pixel shader or other dispatches running in parallel, a smaller thread group can improve performance
MESHLET GENERATION - PART 2

- What is a recommended configuration?
- There are 2 quite common configurations:

  • $V = 128, T = 256$
  • Thread group size = 128
  → Larger meshlets

  • $V = 64, T = 126$ (or 128)
  • Thread group size = 64
  → Smaller meshlets

Use a thread-group-sided stride:

```c
triangles[threadID] = ProcessPrimitive[threadID];
triangles[threadID + 128] = ProcessPrimitive[threadID + 128];
```

- Bonus question: How should you export the primitives?
MESH SHADER - CULLING

- Mesh shader can do triangle/primitive culling
- Might help if the fixed function cull rate is a bottleneck (rasterizer triangle throughput limited)

<table>
<thead>
<tr>
<th>Remove manually from export</th>
<th>SV_CullPrimitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Done before SetMeshOutputCounts</td>
<td></td>
</tr>
<tr>
<td>• No space for vertex attributes is allocated</td>
<td></td>
</tr>
<tr>
<td>• Need to fix export indices via a wave-wide scan</td>
<td>• No effect on SetMeshOutputCounts</td>
</tr>
<tr>
<td>• Space for vertex attributes is still allocated</td>
<td></td>
</tr>
<tr>
<td>• No need to fix any export indices</td>
<td></td>
</tr>
</tbody>
</table>

- Export space is always allocated for the maximum # of vertices and primitives regardless of SetMeshOutputCounts
- In our experiments, SV_CullPrimitive was more beneficial
- The other option might be useful if there are very fat vertices
  • Where applicable, per-primitive attributes could already help
AMPLIFICATION (TASK) SHADER

• Amplification shaders add latency to the render process
• MS calls are executed in the same order as the amplification shader thread groups were launched
• Required to comply with the specified rasterization order

• AS thread groups need to launch enough MS thread groups to hide the latency
• Otherwise, there will be gaps and poor thread utilization

• Typically, AS thread groups should launch at least 32 MS thread groups
• For culling, try to process at least 32 or 64 meshlets per AS thread group
AMPLIFICATION SHADER – DYNAMIC LOD SELECTION

- An amplification shader is called per mesh to select dynamically the LOD and possibly cull

- Issues in above trace:
  - Each mesh only spawns a few MS draws if any (e.g., one AS thread group with only few threads active)
  - A lot of AS thread groups doing very little work and producing very little work
  - Overhead can be estimated by changing AS code to cull all meshlets:
• Payload is stored in group shared memory
• Every thread in the AS thread group can read and write from it
• After the AS thread group finishes, the payload is copied to a ring buffer
  • This copy can be quite slow, every thread copies the entire payload (up to 16KiB)
• Also requires a lot of VGPRs
  • Load from group shared memory into VGPRs
  • Copy from VGPRs to ring buffer using buffer_store instruction
  
→ Use as little payload as possible!
THE FUTURE OF THE GEOMETRY PIPELINE

The Past
- Input Assembler
- Vertex Shader
- Rasterizer
- Pixel Shader

The Present
- Amplification Shader
- Mesh Shader
- Rasterizer
- Pixel Shader

The Future
- Rasterizer
- Pixel Shader
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!

Launch Mesh Shaders from another Shader!
Mesh Shaders are coming to GPU Work Graphs!

Multiple Amplifications
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!

Classification
THE FUTURE OF THE GEOMETRY PIPELINE

Mesh Shaders are coming to GPU Work Graphs!
The Future of the Geometry Pipeline

Mesh Shaders are coming to GPU Work Graphs!
CONCLUSION

- Mesh shaders are a new way to process geometry
- Closer to the underlying hardware
- Allows for more control on how to use the hardware to process the geometry

- Can do things that were previously not possible
  - More opportunities for compression
- Can process any type of primitives, e.g. quads
- Can procedurally generate geometry in an efficient way

- For good performance, pay attention to
  - Meshlet generation
  - Vertex and primitive export
  - AS thread groups launching enough MS thread groups
SPECIAL THANKS

- Łukasz Izdebski
- Dawid Masiukiewicz
- Michael Grossfeld
- Todd Martin
- Dominik Baumeister
- Felix Kawala
- Pirmin Pfeifer
- Quirin Meyer
- Bastian Kuth
- Carsten Faber
The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD products are as set forth in a signed agreement between the parties or in AMD’s Standard Terms and Conditions of Sale. GD-18

THIS INFORMATION IS PROVIDED ‘AS IS.’ AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


Performance testing (slide 39) done on following system: AMD Ryzen™ 7 5800X, AMD Radeon™ RX 7900 XTX, AsusTeK TUF Gaming X570-Plus, 32GB DDR4-3600, Windows 10 Home 22H2, AMD Software: Adrenalin Edition 24.2.1

© 2024 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Radeon, RDNA, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Vulkan and the Vulkan logo are registered trademarks of the Khronos Group Inc. DirectX is a registered trademark of Microsoft Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective owners.
AMD

together we advance