Lecture 14:
Scheduling the Graphics Pipeline on a GPU

Visual Computing Systems
Stanford CS348K, Spring 2020
Today

- Real-time 3D graphics workload metrics
- Scheduling the graphics pipeline on a modern GPU
Quick Review
Real-time graphics pipeline architecture
# Programming the graphics pipeline

- Issue draw commands

\[\rightarrow\] output image contents change

<table>
<thead>
<tr>
<th>Command Type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>State change</td>
<td>Bind shaders, textures, uniforms</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 1</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new uniforms</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 2</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new shader</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 3</td>
</tr>
<tr>
<td>State change</td>
<td>Change depth test function</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new shader</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 4</td>
</tr>
</tbody>
</table>

Final rendered output should be consistent with the results of executing these commands in the order they are issued to the graphics pipeline.
GPU: heterogeneous parallel processor

We’re now going to talk about this scheduler.
Graphics workload metrics
Let’s consider different workloads

Average triangle size

Image credit:
http://www.mobygames.com/game/android/ghostbusters-slime-city/screenshots/gameShotId,852293/
Triangle size
(data from 2010)

[source: NVIDIA]
Low geometric detail
Surface tessellation

Procedurally generate fine triangle mesh from coarse mesh representation

Coarse geometry

Post-Tessellation (fine) geometry

[Image credit: Loop et al. 2009]
Graphics pipeline with tessellation

Five programmable stages in modern pipeline (OpenGL 4, Direct3D 11)

Vertices
- 1 in / 1 out

Primitives
- 3 in / 1 out (for tris)
- 1 in / small N out

Fragments
- 1 in / N out

Pixels
- 1 in / 1 out

Vertex Generation

Vertex Processing

Primitive Generation

Primitive Processing

Rasterization (Fragment Generation)

Fragment Processing

Frame-Buffer Ops

Coarse Vertices
- 1 in / 1 out

Coarse Primitives
- 1 in / 1 out

Fine Vertices
- 1 in / 1 out

Fine Primitives
- 1 in / small N out

Fine Primitive Generation (for tris)

Tessellation

Fine Primitive Processing

Rasterization (Fragment Generation)

Fragment Processing

Frame-Buffer Ops
Scene depth complexity

Rough approximation: $TA = SD$

$T = \# \text{ triangles}$

$A = \text{average triangle area}$

$S = \text{pixels on screen}$

$D = \text{average depth complexity}$
"Diamond" structure of graphics workload

Intermediate data streams tend to be larger than scene inputs or image output

Amount of data generated (size of stream between consecutive stages)

Compact geometric model

High-resolution (post tessellation) mesh

Frame buffer pixels

1 in / 1 out

3 in / 1 out (for tris)

1 in / small N out

1 in / 1 out

1 in / N out

1 in / 1 out

1 in / N out

1 in / 1 out

1 in / 1 out

1 in / 0 or 1 out

Vertex Generation

Coarse Vertices

Coarse Primitives

Fine Vertices

Coarse Primitive Processing

Tessellation

Fine Vertex Processing

Fine Primitive Generation

Fine Primitive Processing

Rasterization (Fragment Generation)

Fragment Processing

Frame-Buffer Ops
Key 3D graphics workload metrics

- Data amplification from stage to stage
  - Average triangle size (amplification in rasterizer: 1 triangle -> N pixels)
  - Expansion during primitive processing (if enabled)
  - Tessellation factor (if tessellation enabled)

- [Vertex/fragment/geometry] shader cost
  - How many instructions?
  - Ratio of math to data access instructions?

- Scene depth complexity
  - Determines number of depth and color buffer writes
Graphics pipeline workload changes dramatically across draw commands

- Triangle size is scene and frame dependent
  - Move far away from an object, triangles get smaller
  - Vary within a frame (characters are usually higher resolution meshes than buildings)

- Varying complexity of materials, different number of lights illuminating surfaces
  - Tens to a few hundreds of instructions per shader

- Stages can be disabled
  - Depth-only rendering = NULL fragment shader
  - Post-processing effects = no vertex work

- Thousands of state changes and draw calls per frame

Example: rendering a “depth map” requires vertex shading but no fragment shading
Parallelizing the graphics pipeline

Adopted from slides by Kurt Akeley and Pat Hanrahan (Stanford CS448 Spring 2007)
GPU: heterogeneous parallel processor

We're now going to talk about this scheduler.
Reminder: requirements + workload challenges

- Pipeline accepts sequence of commands
  - Draw commands
  - State modification commands

- Processing commands has sequential semantics
  - Effects of command A must be visible before those of command B

- Relative cost of pipeline stages changes frequently and unpredictably (e.g., due to changing triangle size, rendering mode)

- Ample opportunities for parallelism
  - Many triangles, vertices, fragments, etc.
Simplified pipeline

For now: just consider all geometry processing work (vertex/primitive processing, tessellation, etc.) as “geometry” processing.

(I’m drawing the pipeline this way to match the suggested readings under this lecture)
Simple parallelization (pipeline parallelism)

Separate hardware unit is responsible for executing work in each stage

What is my maximum speedup?
A cartoon GPU:
Assume we have four separate processing pipelines
Leverages data-parallelism present in rendering computation
More realistic GPU

- A set of programmable cores (run vertex and fragment shader programs)
- Hardware for rasterization, texture mapping, and frame-buffer access
Molnar’s “sorting” taxonomy

Implementations characterized by where communication occurs in pipeline

Sort first
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Sort middle
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Sort last fragment
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Sort last image composition
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Output image

Note: The term “sort” can be misleading for some. It may be helpful to instead consider the term “distribution” rather than sort. The implementations are characterized by how and when they redistribute work onto processors.

* The origin of the term sort was from “A Characterization of Ten Hidden-Surface Algorithms”. Sutherland et al. 1974
Sort first
Assign each replicated pipeline responsibility for a region of the output image
Do minimal amount of work (compute screen-space vertex positions of triangle) to determine which region(s) each input primitive overlaps
Sort first work partitioning
(partition the primitives to parallel units based on screen overlap)
Sort first

- Good:
  - Simple parallelization: just replicate rendering pipeline and operate independently in screen regions (order maintained in each)
  - More parallelism = more performance
  - Small amount of sync/communication (communicate original triangles)
  - Early fine occlusion cull ("early z") just as easy as single pipeline
Sort first

- Potential for workload imbalance (one part of screen contains most of scene)
- Extra cost of triangle “pre-transformation” (needed to sort)
- “Tile spread”: as screen tiles get smaller, primitives cover more tiles (duplicate geometry processing across multiple parallel pipelines)
Sort first examples

- **WireGL/Chromium** (parallel rendering with a cluster of GPUs)
  - “Front-end” node sorts primitives to machines
  - Each GPU is a full rendering pipeline (responsible for part of screen)

- **Initial parallel versions of Pixar’s RenderMan**
  - Multi-core software renderer
  - Sort surfaces into screen tiles prior to tessellation

* Chromium can also be configured as a sort-last image composition system
Sort middle
Distribute primitives to pipelines (e.g., round-robin distribution)

Assign each rasterizer a region of the render target

Sort after geometry processing based on screen space projection of primitive vertices
Interleaved mapping of screen

- Decrease chance of one rasterizer processing most of scene
- Most triangles overlap multiple screen regions (often overlap all)

Interleaved mapping

Tiled mapping
Fragment interleaving in NVIDIA Fermi

**Fine granularity interleaving**

**Coarse granularity interleaving**

**Question 1:** what are the benefits/weaknesses of each interleaving?
**Question 2:** notice anything interesting about these patterns?
Sort middle interleaved

- Good:
  - Workload balance: both for geometry work AND onto rasterizers (due to interleaving)
  - Does not duplicate geometry processing for each overlapped screen region
Sort middle interleaved

- Bad:
  - Bandwidth scaling: sort is implemented as a broadcast (each triangle goes to many/all rasterizers because of interleaved screen mapping)
  - If tessellation is enabled, must communicate many more primitives than sort first
  - Duplicated per triangle work across rasterizers
SGI RealityEngine

Sort-middle interleaved design

[Akeley 93]
Sort-middle tiled (a.k.a. “chunking”, “bucketing”, “binning”)  

Step 1: sort triangles into bins  
- Divide screen into tiles, one triangle list per “tile” of screen (called a “bin”)  
- Core runs vertex processing, computes 2D triangle/screen-tile overlap, inserts triangle into appropriate bin(s)
Sort-middle interleaved vs. binning

### Interleaved (static) assignment of screen tiles to processors

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

### Assignment to bins

List of bins is a work queue. Bins are dynamically assigned to processors.

<table>
<thead>
<tr>
<th></th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
<td>B5</td>
<td>B6</td>
</tr>
<tr>
<td>B6</td>
<td>B7</td>
<td>B8</td>
<td>B9</td>
<td>B10</td>
<td>B11</td>
<td>B12</td>
</tr>
<tr>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td>B16</td>
<td>B17</td>
<td>B18</td>
</tr>
<tr>
<td>B18</td>
<td>B19</td>
<td>B20</td>
<td>B21</td>
<td>B22</td>
<td>B23</td>
<td></td>
</tr>
</tbody>
</table>
Step 2: per-tile processing

- Cores process bins in parallel, performing rasterization fragment shading and frame buffer update

- While there are more bins to process:
  - Assign bin to available core
  - For all triangles:
    - Rasterize
    - Fragment shade
    - Depth test
    - Update frame buffer

final pixels for NxN tile of render target
Reminder: reading less data conserves power

- Goal: redesign algorithms so that they make good use of on-chip memory or processor caches
  - And therefore transfer less data from memory

- A fact you might not have heard:
  - It is far more costly (in energy) to load/store data from memory, than it is to perform an arithmetic operation

  "Ballpark" numbers
  - Integer op: \( \sim 1 \) pJ *
  - Floating point op: \( \sim 20 \) pJ *
  - Reading 64 bits from small local SRAM (1mm away on chip): \( \sim 26 \) pJ
  - Reading 64 bits from low power mobile DRAM (LPDDR): \( \sim 1200 \) pJ

Implications
  - Reading 10 GB/sec from memory: \( \sim 1.6 \) watts

* Cost to just perform the logical operation, not counting overhead of instruction decode, load data from registers, etc. [Sources: Bill Dally (NVIDIA), Tom Olson (ARM)]
What should the screen size of the bins be?

- Small enough for a tile of the color buffer and depth buffer (potentially supersampled) to fit in a shader processor core’s on-chip storage (i.e., cache)

- Tile sizes in range 16x16 to 64x64 pixels are common

- ARM Mali GPU: commonly uses 16x16 pixel tiles
Tiled rendering “sorts” the scene in 2D space to enable efficient color/depth buffer access

Consider rendering without a sort: (process triangles in order given)

This sample updated three times, but may have fallen out of cache in between accesses

Now consider step 2 of a tiled renderer:

- Initialize Z and color buffer for tile
  - for all triangles in tile:
    - for all each fragment:
      - shade fragment
      - update depth/color
    - write color tile to final image buffer

Q. Why doesn’t the renderer need to write depth buffer in memory? *

Q. Why doesn’t the renderer need to read color or depth buffer from memory?

* Assuming application does not need depth buffer for other purposes.
Tile-based deferred rendering (TBDR)

- Many mobile GPUs implement deferred shading in the hardware!
- Divide step 2 of tiled pipeline into two phases:
  - Phase 1: compute what fragment is visible at every sample
  - Phase 2: perform shading of only the visible quad fragments

![Diagram of TBDR process]

Stanford CS348K, Spring 2020
Sort middle tiled (chunked)

- **Good:**
  - Good load balance (distribute many buckets onto rasterizers)
  - Low bandwidth requirements *(why? when?)*
  - Challenge: “bucketing” sort has low contention (assuming each triangle only touches a small number of buckets), but there still is contention

- **Recent examples:**
  - Many mobile GPUs: Imagination PowerVR, ARM Mali, Qualcomm Adreno
  - Parallel software rasterizers
    - Intel Larrabee software rasterizer
    - NVIDIA CUDA software rasterizer
Sort last
Sort last fragment

Distribute primitives to top of pipelines (e.g., round robin)
Sort after fragment processing based on (x,y) position of fragment
Sort last fragment

- Good:
  - No redundant geometry processing or rasterization (but early z-cull is a problem)
  - Point-to-point communication during sort
  - Interleaved pixel mapping results in good workload balance for frame-buffer ops
**Sort last fragment**

- **Bad:**
  - Pipelines may stall due to primitives of varying size (due to order requirement)
  - Bandwidth scaling: many more fragments than triangles
  - Hard to implement early occlusion cull (more bandwidth challenges)
Sort last image composition

Each pipeline renders some fraction of the geometry in the scene. Combine the color buffers, according to depth, into the final image.
Sort last image composition

Other combiners possible
Sort last image composition

- Breaks graphics pipeline architecture abstraction: cannot maintain pipeline’s sequential semantics

- Simple implementation: N separate rendering pipelines
  - Can use off-the-shelf GPUs to build a massive rendering system
  - Coarse-grained communication (image buffers)

- Similar load imbalance problems as sort-last fragment

- Under high depth complexity, bandwidth requirement is lower than sort last fragment
  - Communicate final pixels, not all fragments
Recall: modern OpenGL 4 / Direct3D 11 pipeline

Five programmable stages
Modern GPU: programmable parts of pipeline virtualized on pool of programmable cores

Hardware is a **heterogeneous** collection of resources (programmable and non-programmable)

Programmable resources are time-shared by vertex/primitive/fragment processing work
Must keep programmable cores busy: sort everywhere
Hardware work distributor assigns work to cores (based on contents of inter-stage queues)
Sort everywhere

(How modern high-end GPUs are scheduled)
Sort everywhere

Distribute primitives to top of pipelines
Redistribute after geometry processing (e.g., round robin)
Sort after fragment processing based on (x,y) position of fragment
Implementing sort everywhere

(Challenge: rebalancing work at multiple places in the graphics pipeline to achieve efficient parallel execution, while maintaining triangle draw order)
Starting state: draw commands enqueued for pipeline

Input: three triangles to draw
(fragments to be generated for each triangle by rasterization are shown below)

Assume batch size is 2 for assignment to rasterizers.
After geometry processing, first two processed triangles assigned to rast 0

Input:

Assume batch size is 2 for assignment to rasterizers.
Assign next triangle to rast 1 (round robin policy, batch size = 2)

Q. What is the ‘next’ token for?

Input:

Draw

Draw

Draw

Interleaved render target
Rast 0 and rast 1 can process T1 and T3 simultaneously
(Shaded fragments enqueued in frame-buffer unit input queues)

Input:

Draw \( T_1 \) → 1 2 3 4
Draw \( T_2 \) → 1 2 3 4
Draw \( T_3 \) → 1 2 3

Interleaved render target
FB 0 and FB 1 can simultaneously process fragments from rast 0
(Notice updates to frame buffer)
Fragments from T3 cannot be processed yet. Why?

Input:

Draw \( \Delta \)

\( \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \end{array} \)

Interleaved render target
Rast 0 processes T2
(Shaded fragments enqueued in frame-buffer unit input queues)

Input:

Draw $\Delta_{T1}$ $\rightarrow$ 1 2 3 4
Draw $\Delta_{T2}$ $\rightarrow$ 1 2 3 4
Draw $\Delta_{T3}$ $\rightarrow$ 1 2 3

Interleaved render target
Rast 0 broadcasts ‘next’ token to all frame-buffer units
FB 0 and FB 1 can simultaneously process fragments from rast 0
(Notice updates to frame buffer)
Switch token reached: frame-buffer units start processing input from rast 1
FB 0 and FB 1 can simultaneously process fragments from rast 1
(Notice updates to frame buffer)
Extending to parallel geometry units
Starting state: commands enqueued

Input:

```
Draw T1  ➔  1  2  3  4
Draw T2  ➔  1  2  3  4
Draw T3  ➔  1  2  3  4
Draw T4  ➔  1  2
```

Assume batch size is 2 for assignment to geom units and to rasterizers.
Distribute triangles to geom units round-robin (batches of 2)

Input:

Draw \[ \mathbf{T}_1 \] → 1 2 3 4

Draw \[ \mathbf{T}_2 \] → 1 2 3 4

Draw \[ \mathbf{T}_3 \] → 1 2 3 4

Draw \[ \mathbf{T}_4 \] → 1 2

Interleaved render target

Distribute triangles to geom units round-robin (batches of 2)
Geom 0 and geom 1 process triangles in parallel
(Results after T1 processed are shown. Note big triangle T1 broken into multiple work items. [Eldridge et al.])

Input:

Draw $\uparrow$T1 $\rightarrow$ 1 2 3 4
Draw $\uparrow$T2 $\rightarrow$ 1 2 3 4
Draw $\uparrow$T3 $\rightarrow$ 1 2 3 4
Draw $\uparrow$T4 $\rightarrow$ 1 2

Interleaved render target
Geom 0 and geom 1 process triangles in parallel
(Triangles enqueued in rast input queues. Note big triangles broken into multiple work items. [Eldridge et al.])

Input:

Draw \( \Delta T_1 \) \( \rightarrow \) 1 2 3 4

Draw \( \Delta T_2 \) \( \rightarrow \) 1 2 3 4

Draw \( \Delta T_3 \) \( \rightarrow \) 1 2 3 4

Draw \( \Delta T_4 \) \( \rightarrow \) 1 2

Geom 0 and geom 1 process triangles in parallel.

Triangles enqueued in rast input queues. Note big triangles broken into multiple work items. [Eldridge et al.]

Interleaved render target
Geom 0 broadcasts ‘next’ token to rasterizers

Input:

```
Draw T1   ─ 1 2 3 4
Draw T2   ─ 1 2 3 4
Draw T3   ─ 1 2 3 4
Draw T4   ─ 1 2
```

Interleaved render target
Rast 0 and rast 1 process triangles from geom 0 in parallel
(Shaded fragments enqueued in frame-buffer unit input queues)

Input:

Draw $\triangle T_1$ $\rightarrow$ 1 2 3 4

Draw $\triangle T_2$ $\rightarrow$ 1 2 3 4

Draw $\triangle T_3$ $\rightarrow$ 1 2 3 4

Draw $\triangle T_4$ $\rightarrow$ 1 2

Interleaved render target
Rast 0 broadcasts ‘next’ token to FB units (end of geom 0, rast 0)
Frame-buffer units process frags from (geom 0, rast 0) in parallel

(Notice updates to frame buffer)
"End of rast 0" token reached by FB: FB units start processing input from rast 1 (fragments from geom 0, rast 1)
“End of geom 0” token reached by rast units: rast units start processing input from geom 1 (note “end of geom 0, rast 1” token sent to rast input queues)

Input:

- Draw $\Rightarrow 1\ 2\ 3\ 4$
- Draw $\Rightarrow 5\ 6\ 7$
- Draw $\Rightarrow 1\ 2\ 3\ 4$
- Draw $\Rightarrow 5$
- Draw $\Rightarrow 1\ 2$

Interleaved render target
Rast 0 processes triangles from geom 1
(Note Rast 1 has work to do, but cannot make progress because its output queues are full)
Rast 0 broadcasts “end of geom 1, rast 0” token to frame-buffer units
Frame-buffer units process frags from (geom 0, rast 1) in parallel
(Notice updates to frame buffer. Also notice rast 1 can now make progress since space has become available)
Switch token reached by FB: FB units start processing input from (geom 1, rast 0)

Input:

<table>
<thead>
<tr>
<th>Draw</th>
<th>T1</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw</td>
<td>T2</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Draw</td>
<td>T3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Draw</td>
<td>T4</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Interleaved render target
Frame-buffer units process frags from \((\text{geom 1, rast 0})\) in parallel

(Notice updates to frame buffer)

Input:

Draw \(\Delta T1\) \(\rightarrow\) \(\{1, 2, 3, 4\}\)

Draw \(\Delta T2\) \(\rightarrow\) \(\{1, 2, 3, 4\}\)

Draw \(\Delta T3\) \(\rightarrow\) \(\{1, 2, 3, 4\}\)

Draw \(\Delta T4\) \(\rightarrow\) \(\{1, 2\}\)
Switch token reached by FB: FB units start processing input from (geom 1, rast 1)

Input:

Draw T1 → 1 2 3 4
Draw T2 → 1 2 3 4
Draw T3 → 1 2 3 4
Draw T4 → 1 2

Interleaved render target
Frame-buffer units process frags from (geom 1, rast 1) in parallel
(Notice updates to frame buffer)

Input:

Draw \(\triangle T_1\) \(\rightarrow\) 1 2 3 4

Draw \(\triangle T_2\) \(\rightarrow\) 1 2 3 4

Draw \(\triangle T_3\) \(\rightarrow\) 1 2 3 4

Draw \(\triangle T_4\) \(\rightarrow\) 1 2

Interleaved render target
Parallel scheduling with data amplification
Geometry amplification

- Consider examples of one-to-many stage behavior during geometry processing in the graphics pipeline:

  - Clipping amplifies geometry (clipping can result in multiple output primitives)

  - Tessellation: pipeline permits thousands of vertices to be generated from a single base primitive (challenging to maintain highly parallel execution)

  - Primitive processing ("geometry shader") outputs up to 1024 floats worth of vertices per input primitive
Thought experiment

Assume round-robin distribution of eight primitives to geometry pipelines, one rasterizer unit.
Consider case of large amplification when processing T1

Result: one geometry unit (the one producing outputs from T1) is feeding the entire downstream pipeline
- Serialization of geometry processing: other geometry units are stalled because their output queues are full (they cannot be drained until all work from T1 is completed)
- Underutilization of rest of chip: unlikely that one geometry producer is fast enough to produce pipeline work at a rate that fills resources of rest of GPU.
Thought experiment: design a scheduling strategy for this case

1. Design a solution that is performant when the expected amount of data amplification is low?
2. Design a solution that is performant when the expected amount of data amplification is high.
3. What about a solution that works well for both?

The ideal solution always executes with maximum parallelism (no stalls), and with maximal locality (units read and write to fixed size, on-chip inter-stage buffers), and (of course) preserves order.
Implementation 1: fixed on-chip storage

Approach 1: make on-chip buffers big enough to handle common cases, but tolerate stalls
- Run fast for low amplification (never move output queue data off chip)
- Run very slow under high amplification (serialization of processing due to blocked units). Bad performance cliff.
Implementation 2: worst-case allocation

Approach 2: never block geometry unit: allocate worst-case space in off-chip buffers (stored in DRAM)
- Run slower for low amplification (data goes off chip then read back in by rasterizers)
- No performance cliff for high amplification (still maximum parallelism, data still goes off chip)
- What is overall worst-case buffer allocation if the four geometry units above are Direct3D 11 geometry shaders?

Large, in-memory buffers
Implementation 3: hybrid

Hybrid approach: allocate output buffers on chip, but spill to off-chip, worst-case size buffers under high amplification

- Run fast for low amplification (high parallelism, no memory traffic)
- Less of performance cliff for high amplification (high parallelism, but incurs more memory traffic)
NVIDIA GPU implementation

Optionally resort work after “Hull” shader stage (since amplification factor known)