Lecture 17:

Scheduling the Graphics Pipeline on a GPU

Visual Computing Systems
Stanford CS348V, Winter 2018

Today

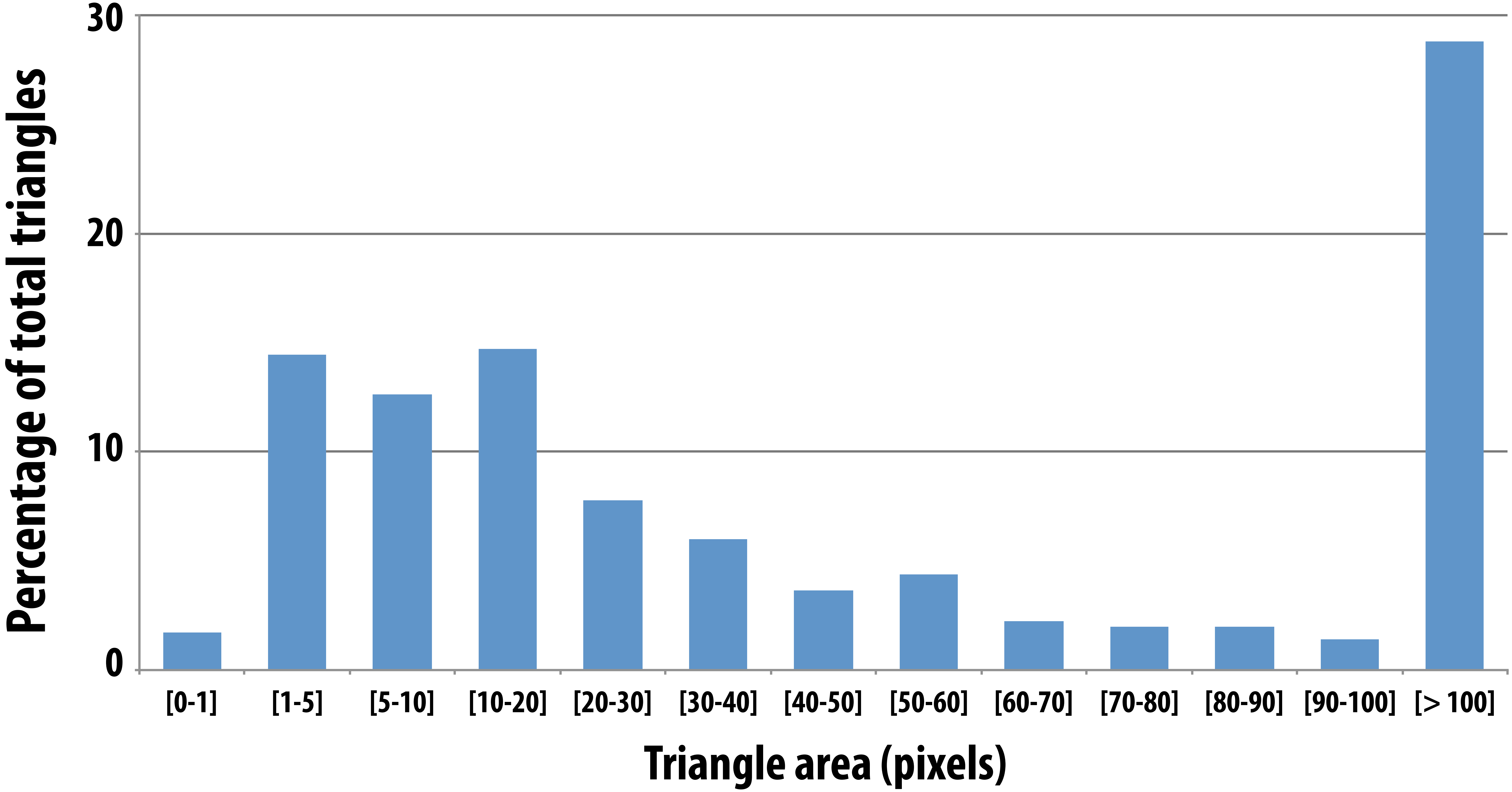
- Real-time 3D graphics workload metrics

- Scheduling the graphics pipeline on a modern GPU

Quick aside: tessellation

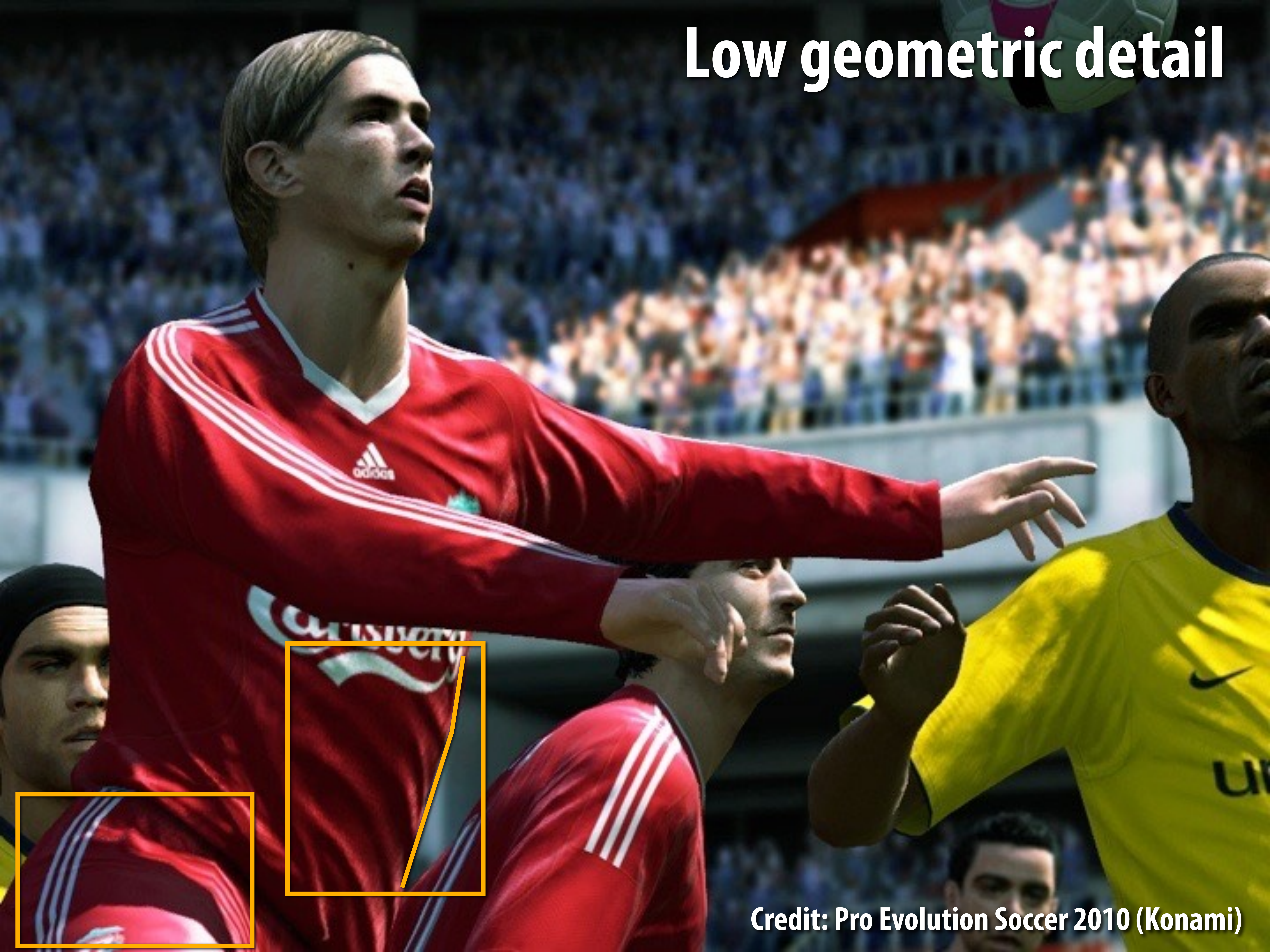
Triangle size

(data from 2010)



[source: NVIDIA]

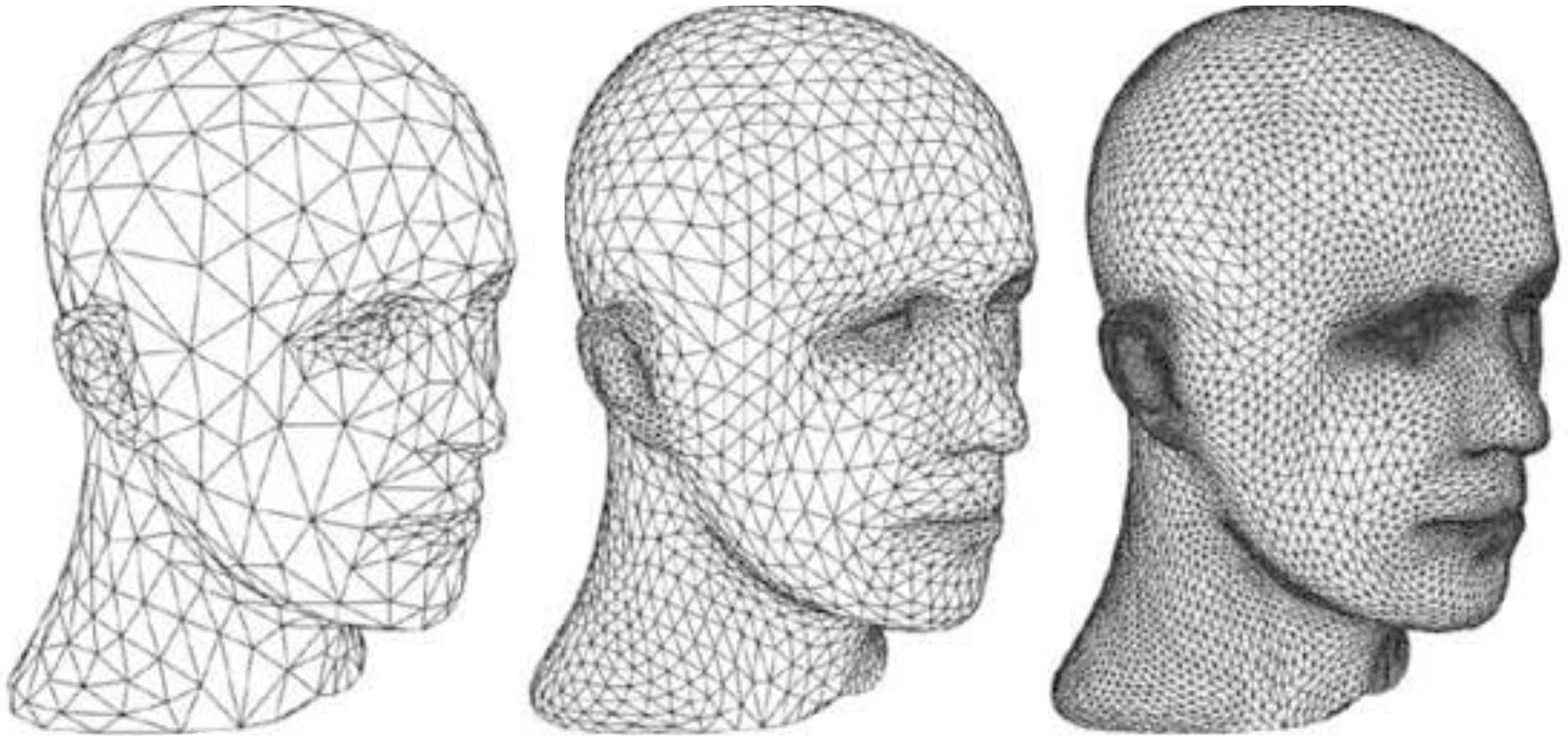
Low geometric detail

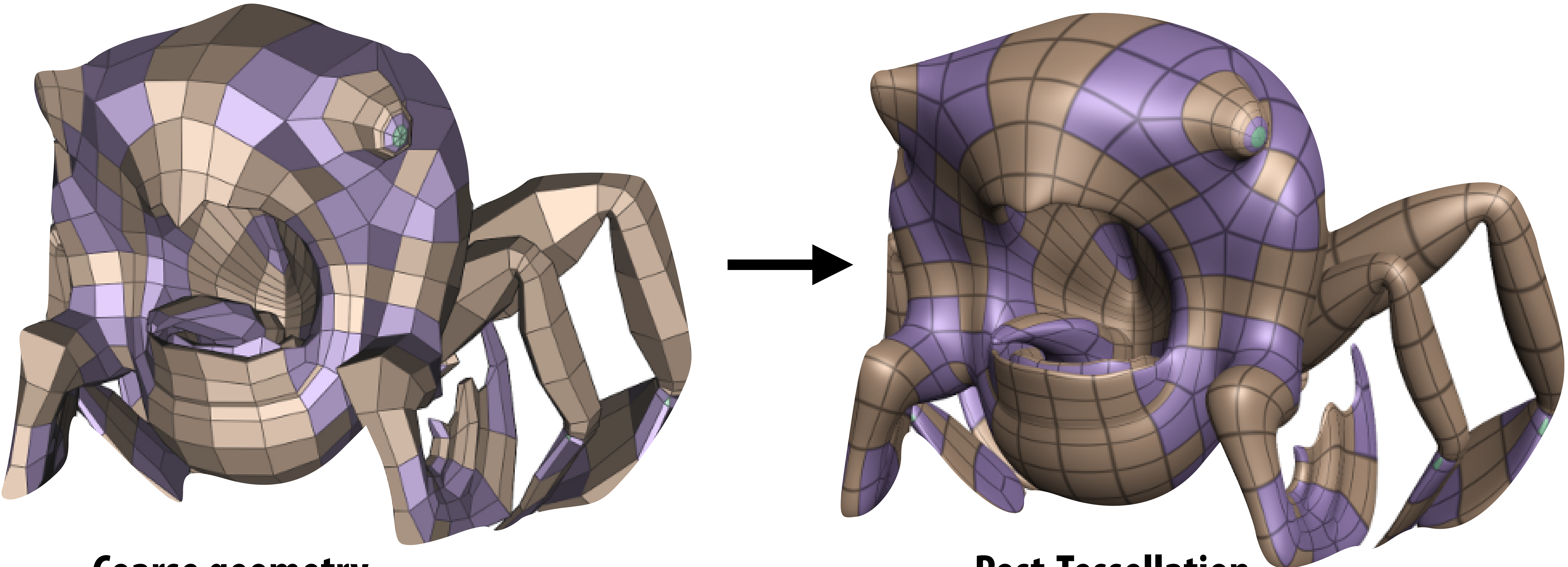


Credit: Pro Evolution Soccer 2010 (Konami)

Surface tessellation

Procedurally generate fine triangle mesh from coarse mesh representation





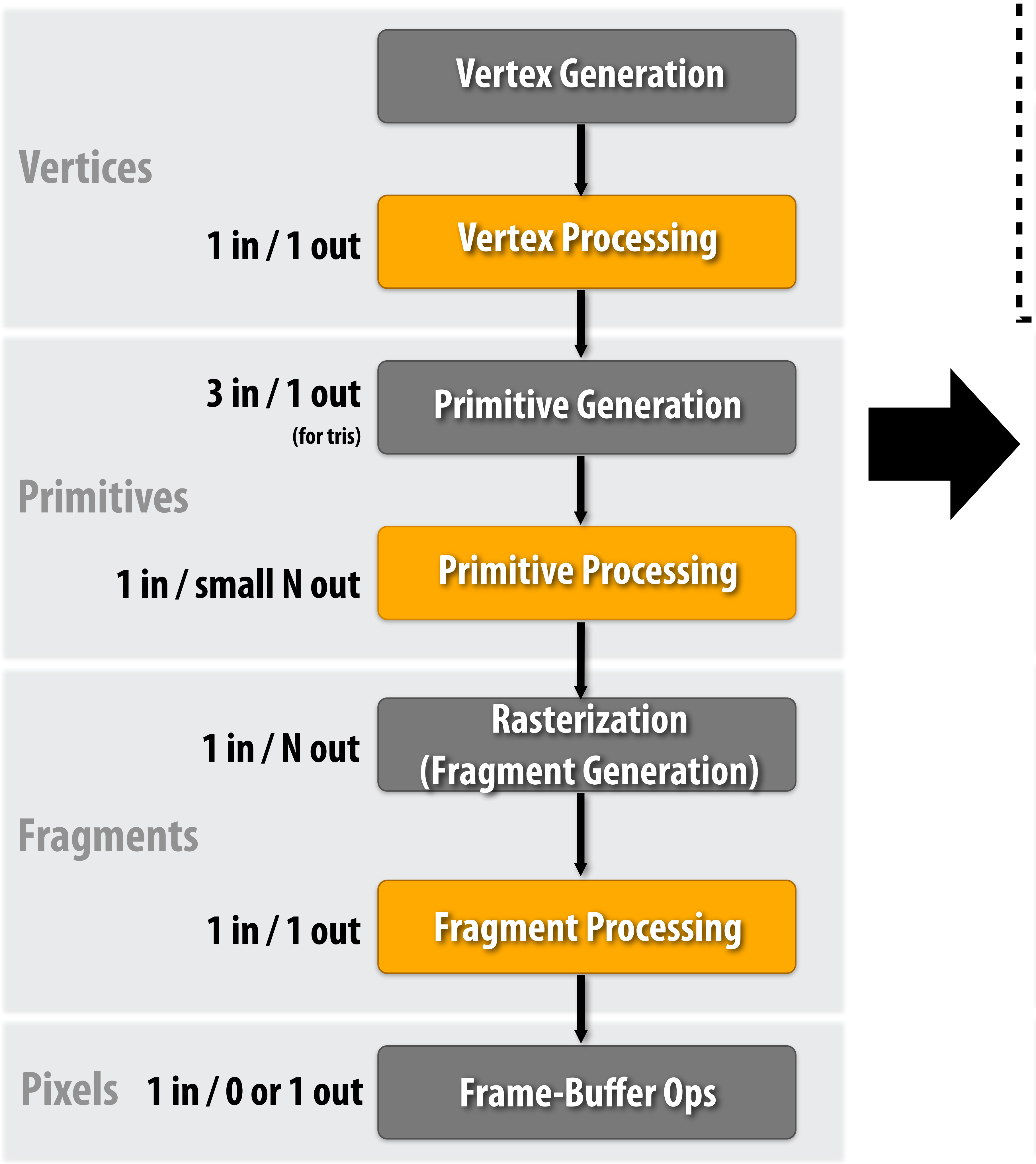
Coarse geometry

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

Vertex Generation

1 in / 1 out — Vertex Processing

Primitives

3 in / 1 out (for tris) — Primitive Generation

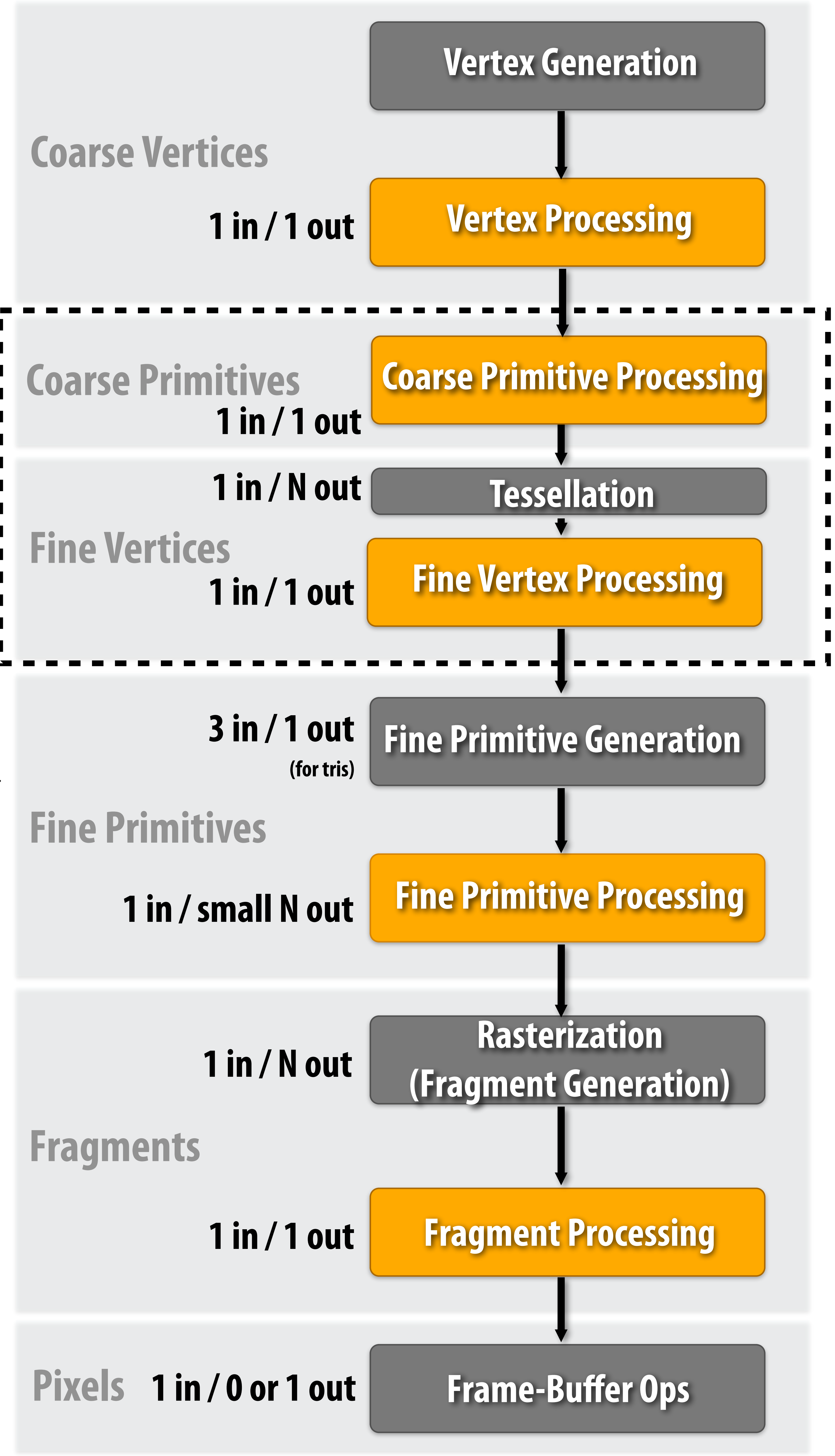
1 in / small N out — Primitive Processing

Fragments

1 in / N out — Rasterization (Fragment Generation)

1 in / 1 out — Fragment Processing

Pixels 1 in / 0 or 1 out — Frame-Buffer Ops



Coarse Vertices

Vertex Generation

1 in / 1 out — Vertex Processing

Coarse Primitives

1 in / 1 out — Coarse Primitive Processing

1 in / N out — Tessellation

Fine Vertices

1 in / 1 out — Fine Vertex Processing

Fine Primitives

3 in / 1 out (for tris) — Fine Primitive Generation

1 in / small N out — Fine Primitive Processing

Fragments

1 in / N out — Rasterization (Fragment Generation)

1 in / 1 out — Fragment Processing

Pixels 1 in / 0 or 1 out — Frame-Buffer Ops

Graphics workload metrics

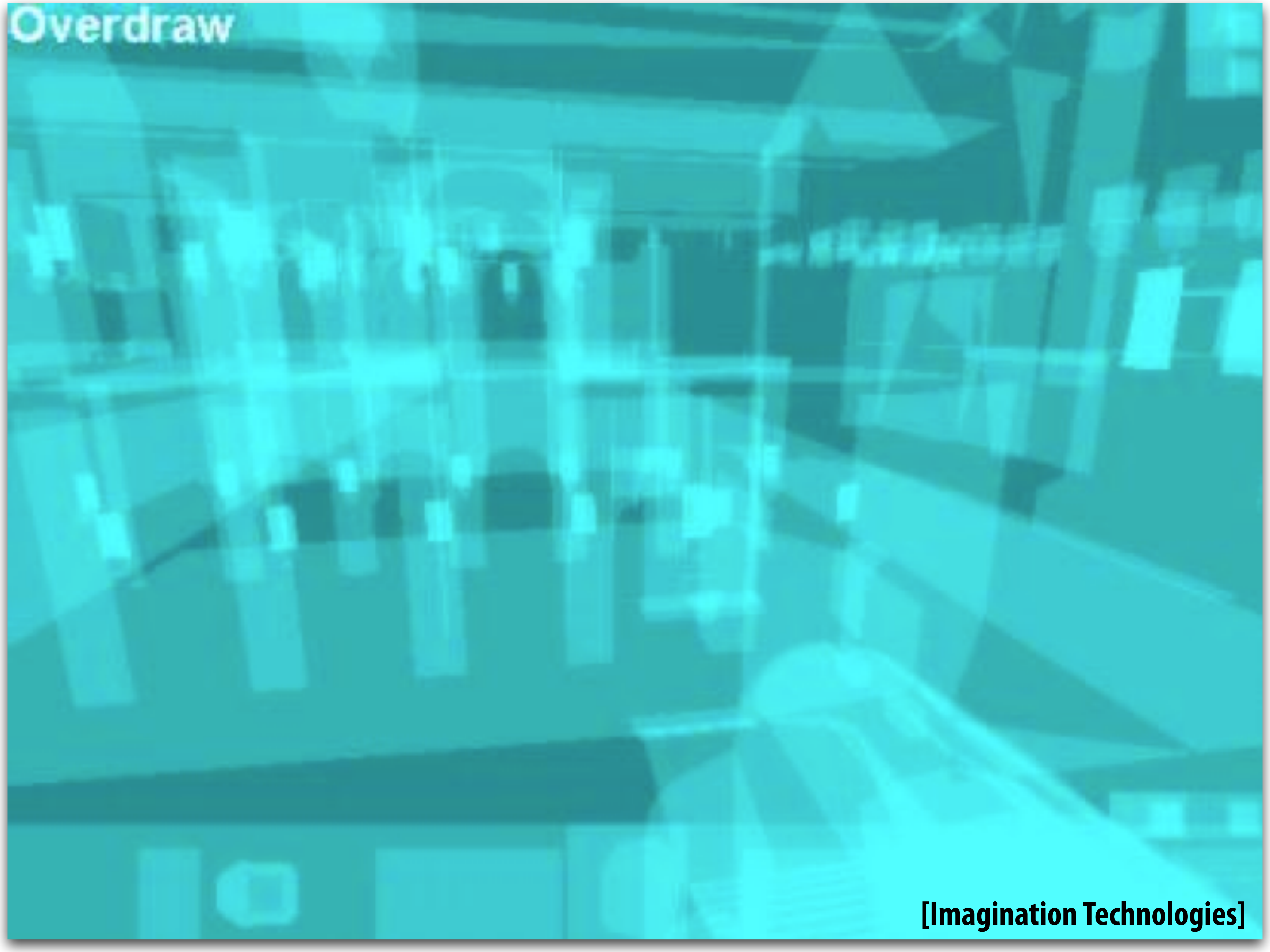
Key 3D graphics workload metrics

- **Data amplification from stage to stage**
  - **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**
  - **Recall: early/high Z-cull optimizations are most efficient when pipeline receives triangles in depth order**

Scene depth complexity



[Imagination Technologies]

Rough approximation: $TA = SD$

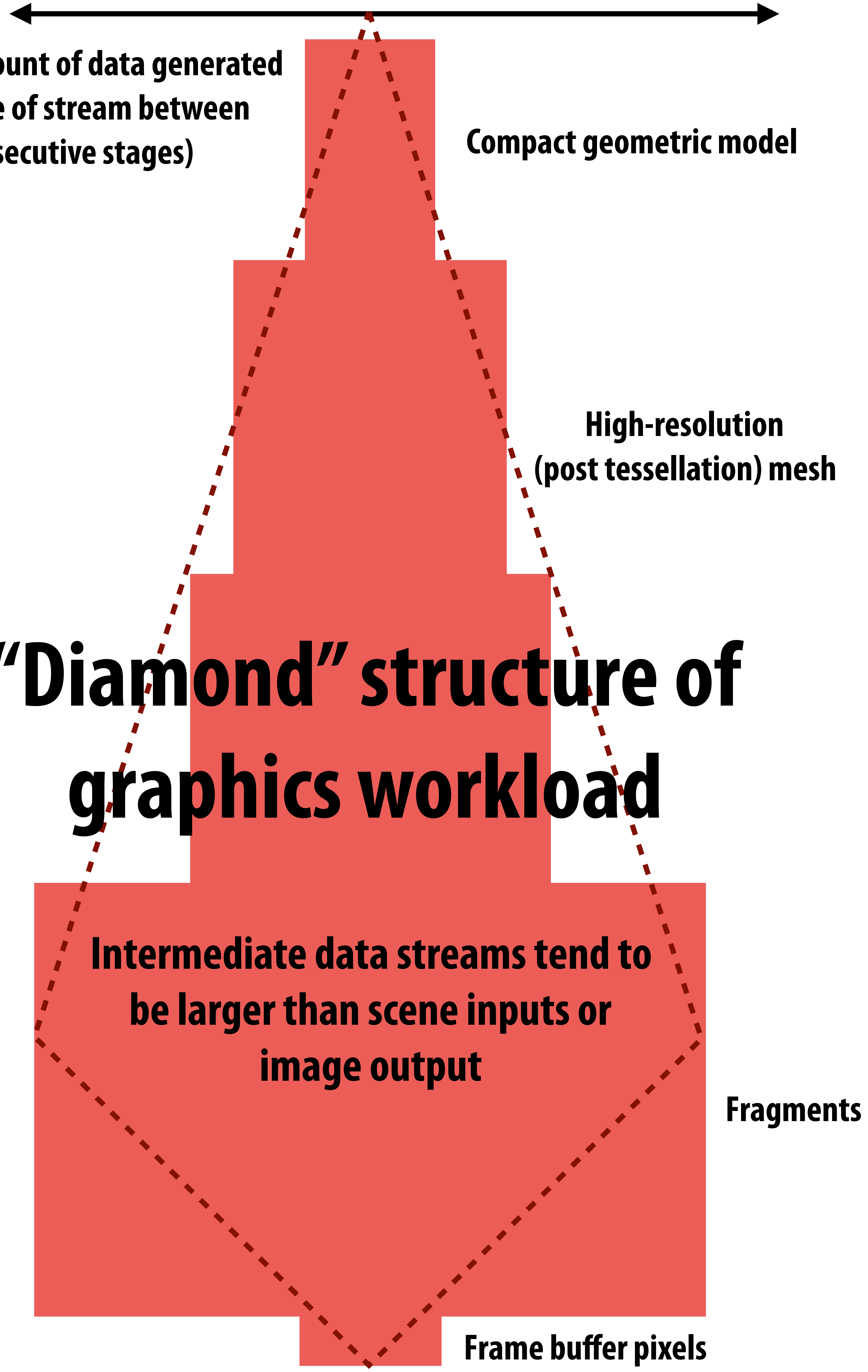
T = # triangles

A = average triangle area

S = pixels on screen

D = average depth complexity

"Diamond" structure of graphics workload



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

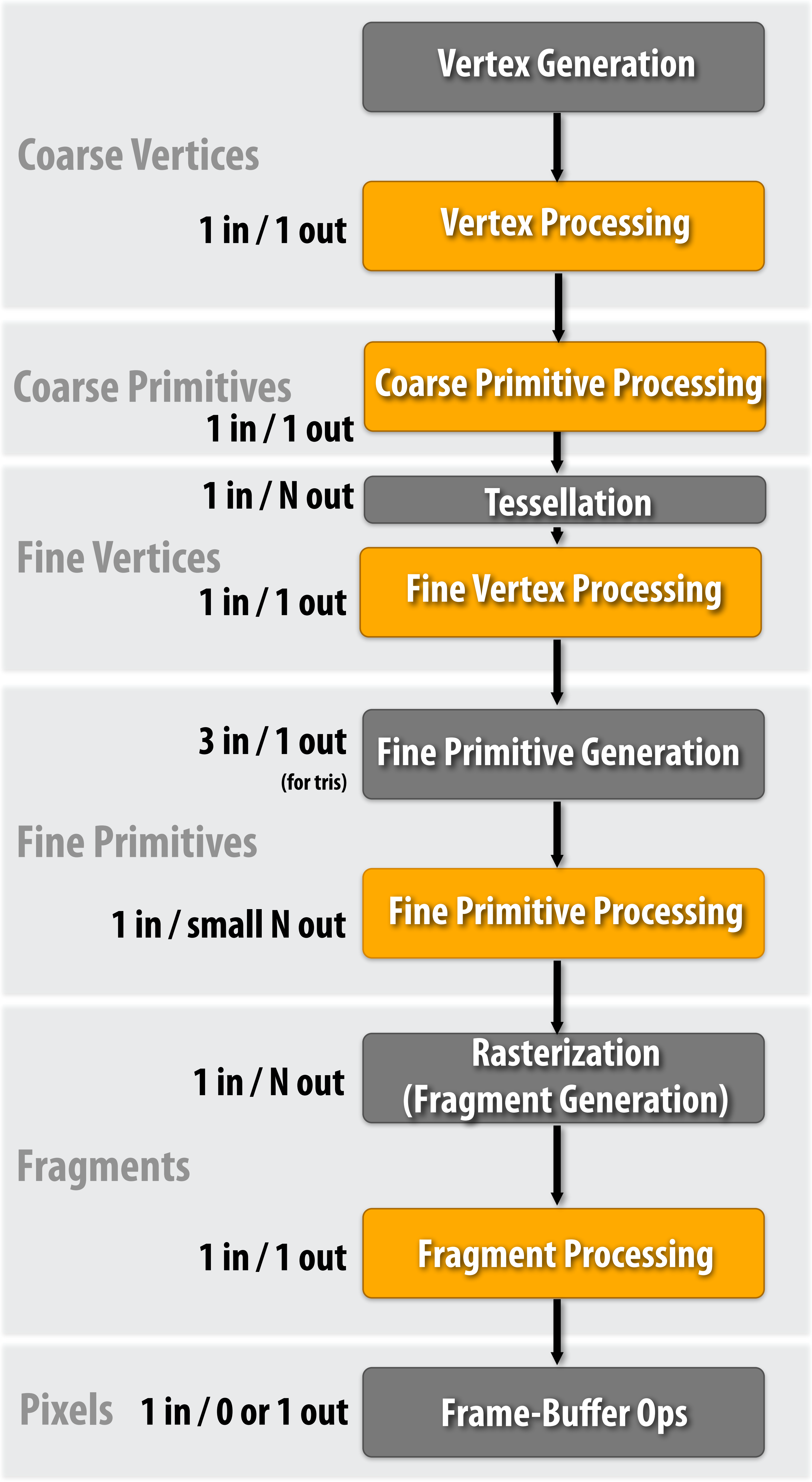
Compact geometric model

High-resolution (post tessellation) mesh

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

Fragments

Frame buffer pixels



Coarse Vertices

Vertex Generation

1 in / 1 out — Vertex Processing

Coarse Primitives

1 in / 1 out — Coarse Primitive Processing

1 in / N out — Tessellation

Fine Vertices

1 in / 1 out — Fine Vertex Processing

3 in / 1 out (for tris) — Fine Primitive Generation

Fine Primitives

1 in / small N out — Fine Primitive Processing

1 in / N out — Rasterization (Fragment Generation)

Fragments

1 in / 1 out — Fragment Processing

Pixels 1 in / 0 or 1 out — Frame-Buffer Ops

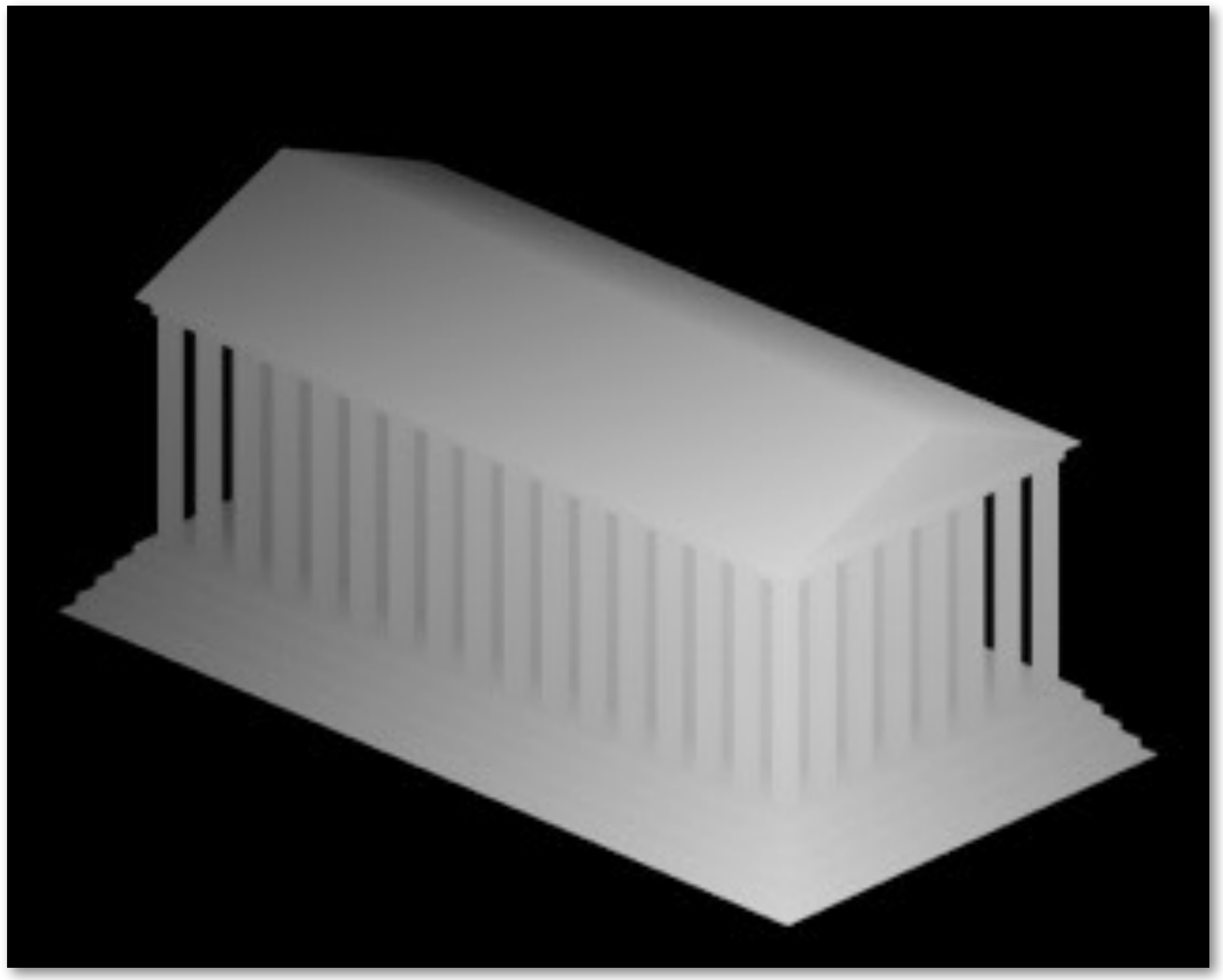
Graphics pipeline workload changes dramatically across draw commands

- **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)

- **Varying complexity of materials, different number of lights illuminating surfaces**
  - No such thing as a "canonical" shader
  - Tens to a few hundreds of instructions per shader

- **Stages can be disabled**
  - Shadow map creation = NULL fragment shader
  - Post-processing effects = no vertex work

- **Thousands of state changes and draw calls per frame**

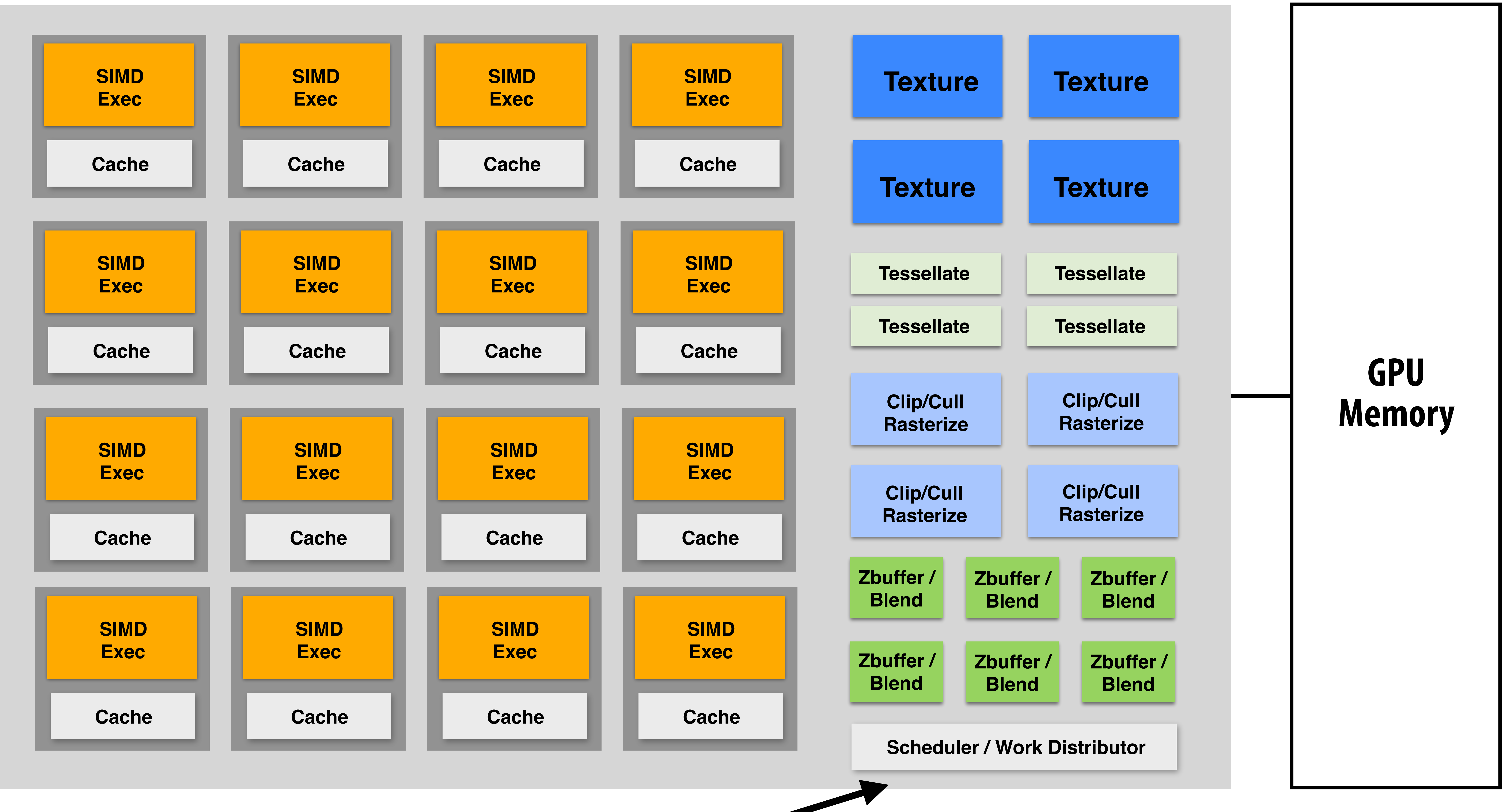


Example: rendering a "depth map" requires vertex shading but no fragment shading

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

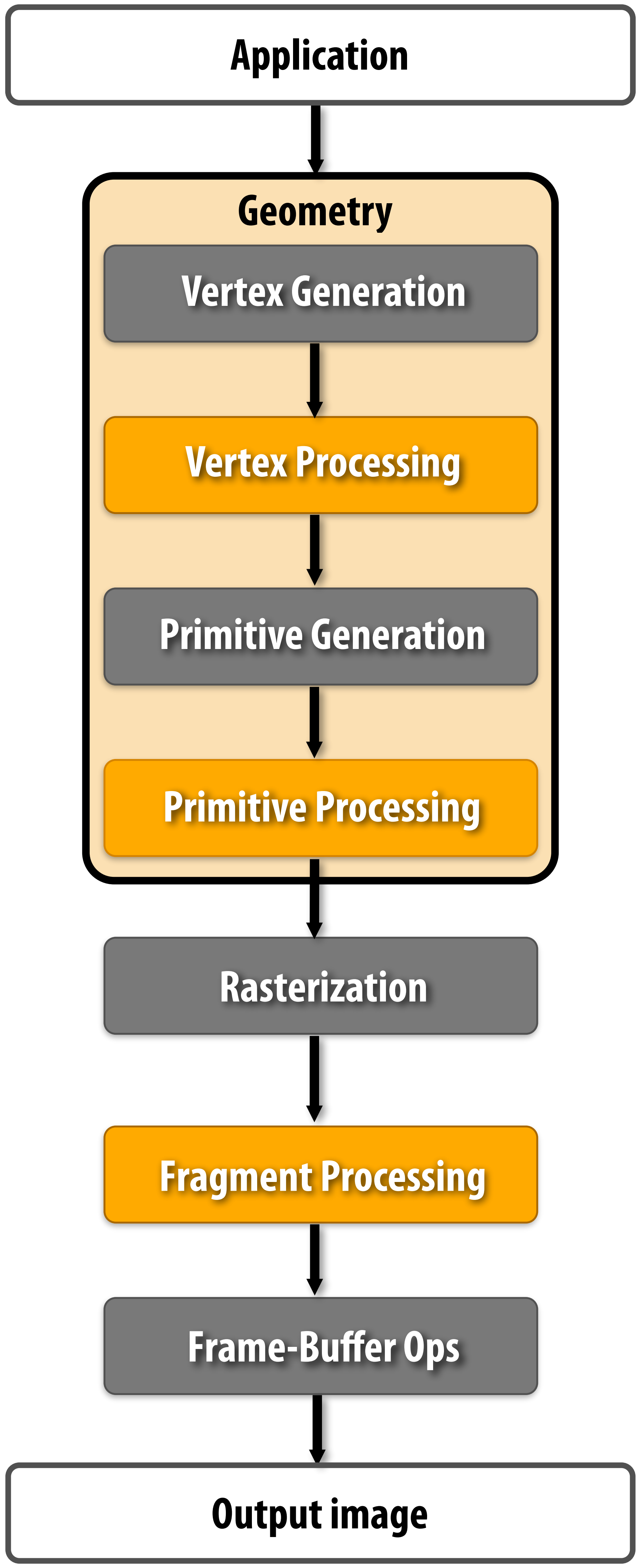
- Relative cost of pipeline stages changes frequently and unpredictably (e.g., due to changing triangle size, rendering mode)

- 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.

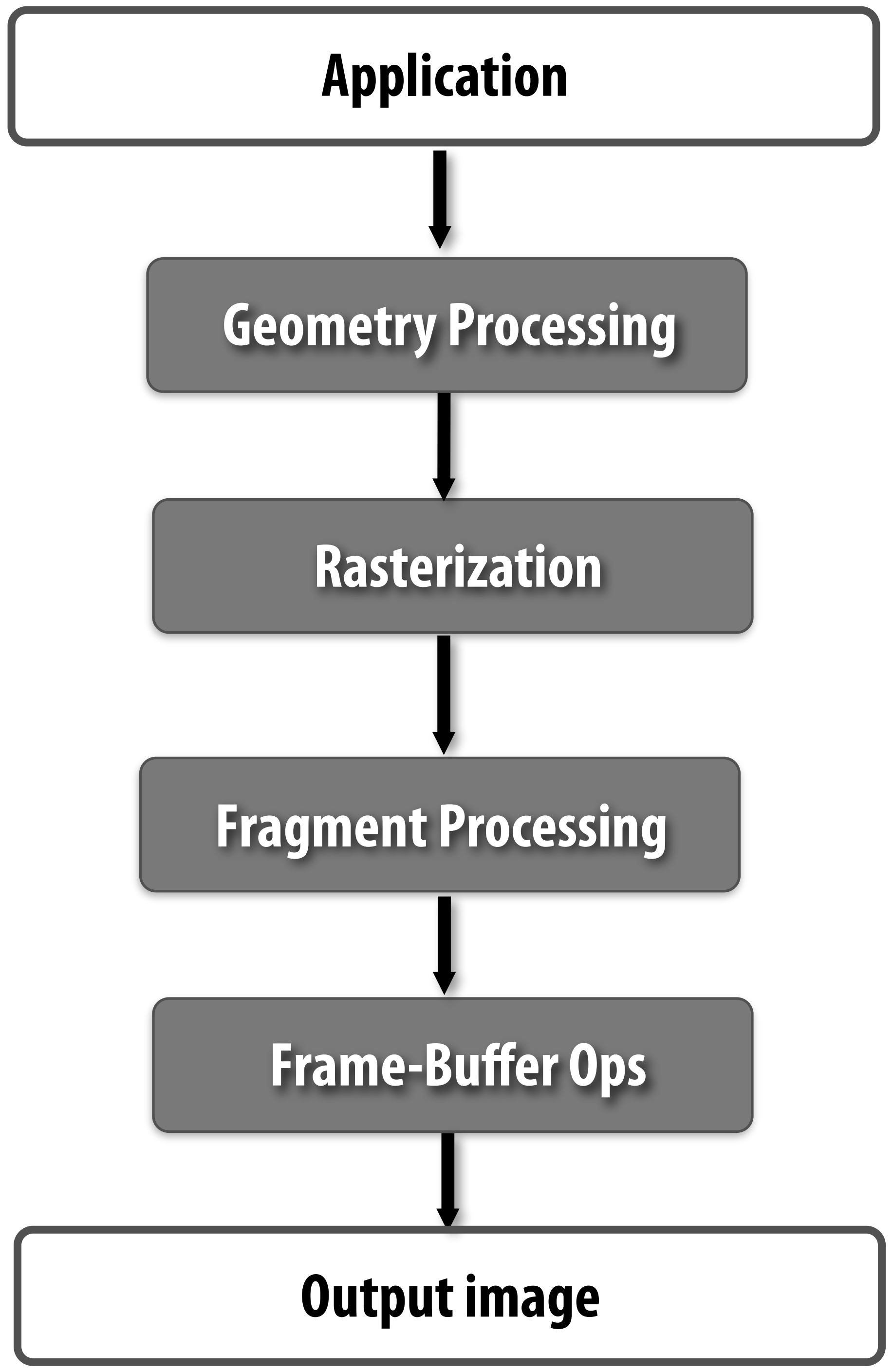
(I’m drawing the pipeline this way to match tonight’s suggested readings)



Simple parallelization (pipeline parallelism)

Separate hardware unit is responsible for executing work in each stage

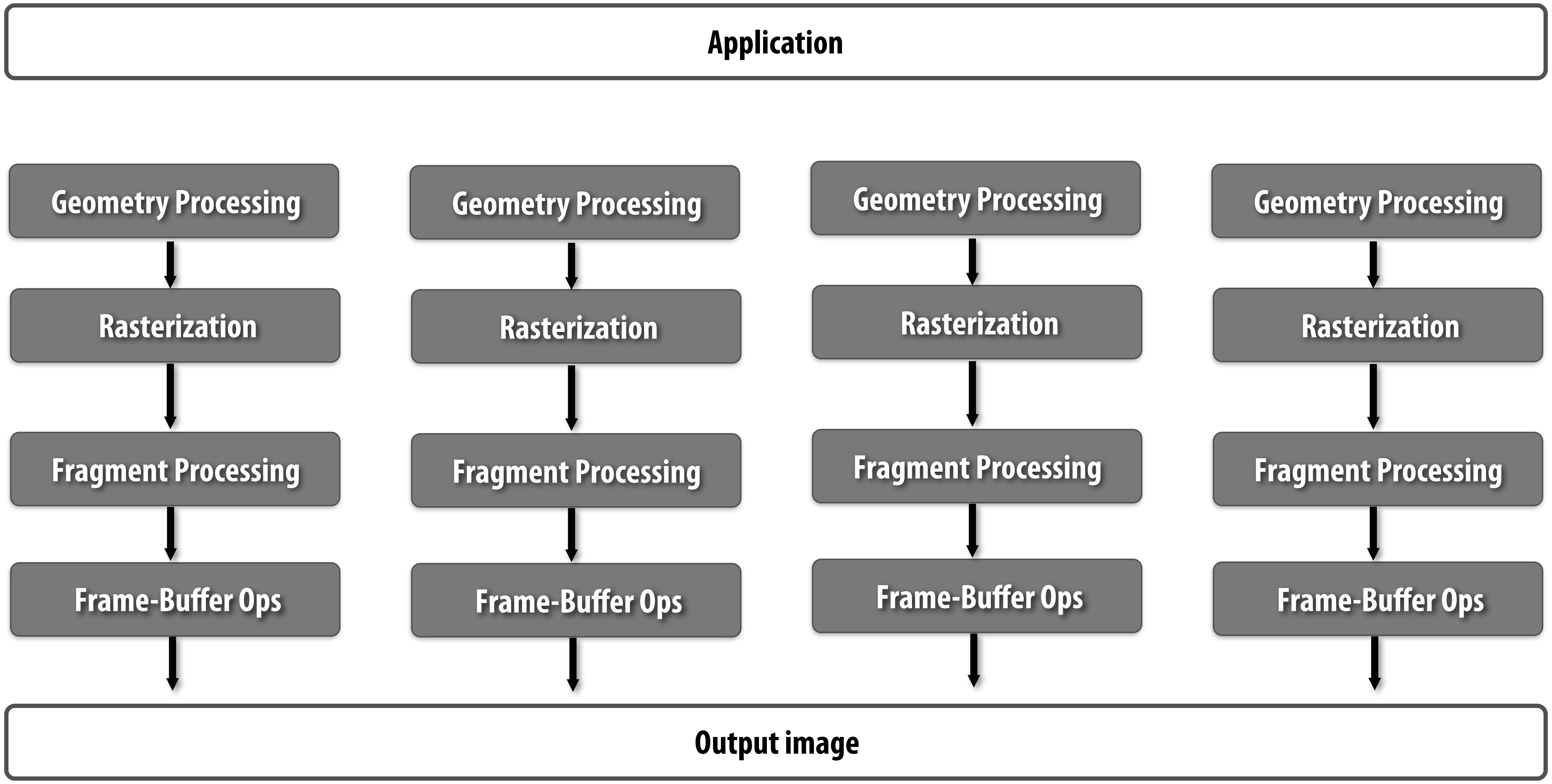
What is my maximum speedup?



A cartoon GPU:

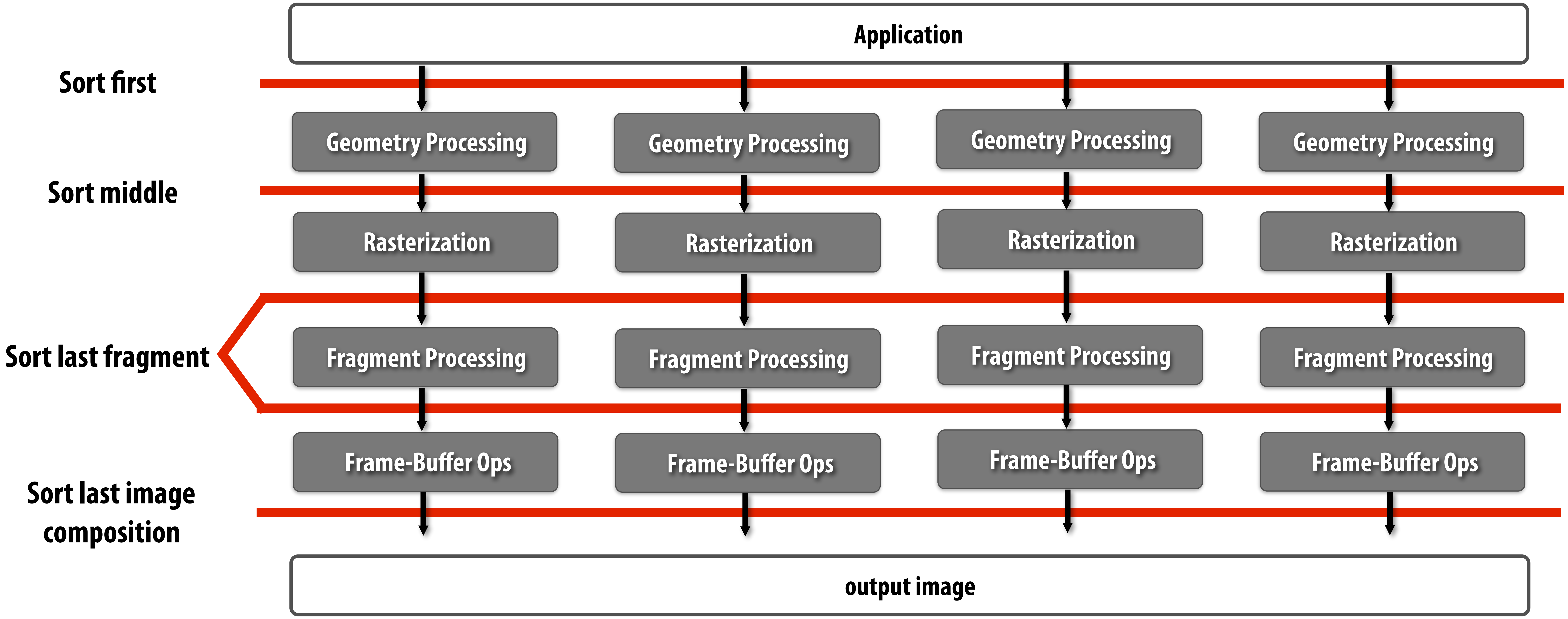
Assume we have four separate processing pipelines

Leverages data-parallelism present in rendering computation



Molnar’s sorting taxonomy

Implementations characterized by where communication occurs in pipeline

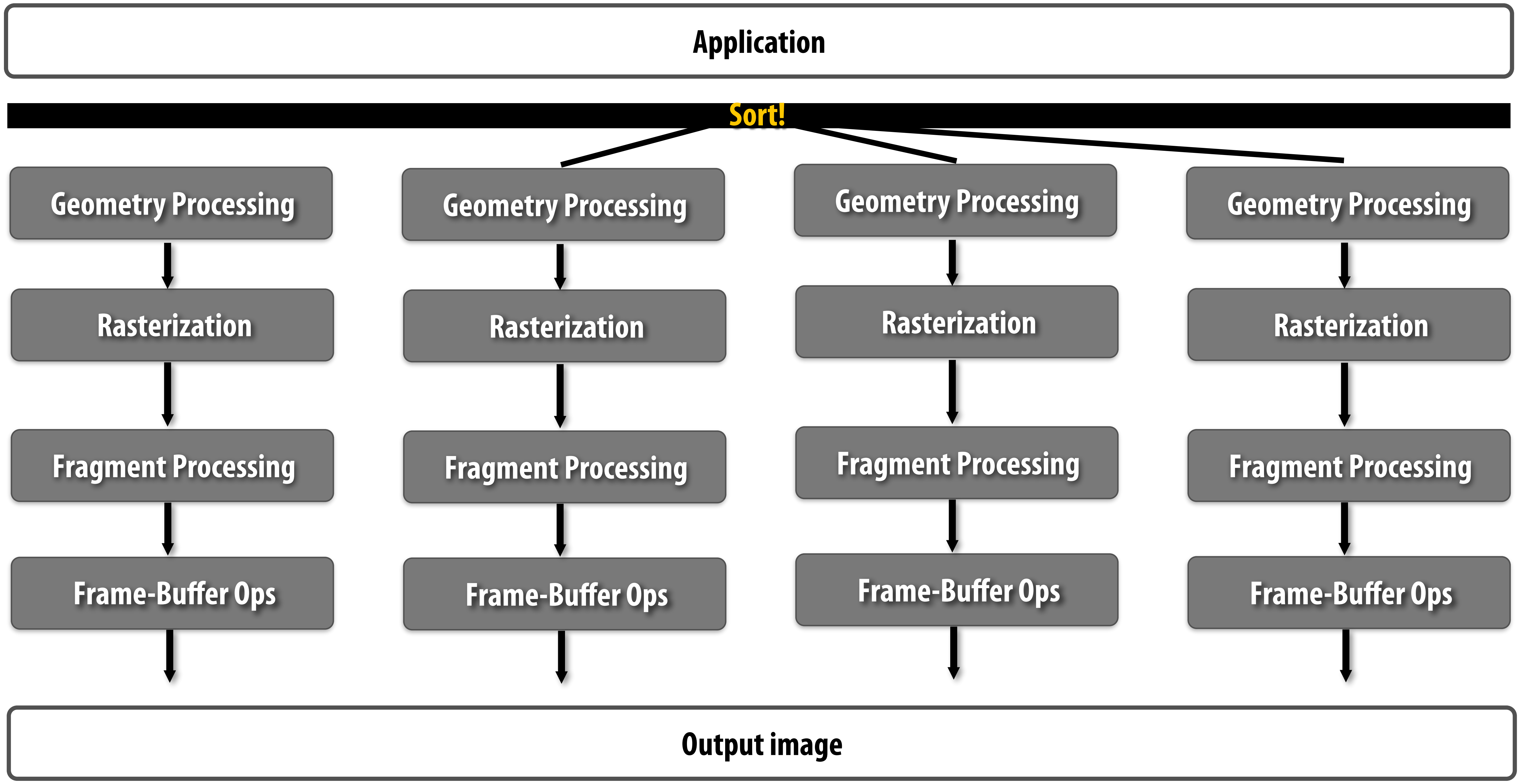


Note: The term “sort” can be misleading for some. It may be helpful to instead consider the term “distribution” rather than sort. The implementations are characterized by how and when they redistribute work onto processors. *

* The origin of the term sort was from “A Characterization of Ten Hidden-Surface Algorithms”. Sutherland et al. 1974

Sort first

Sort first

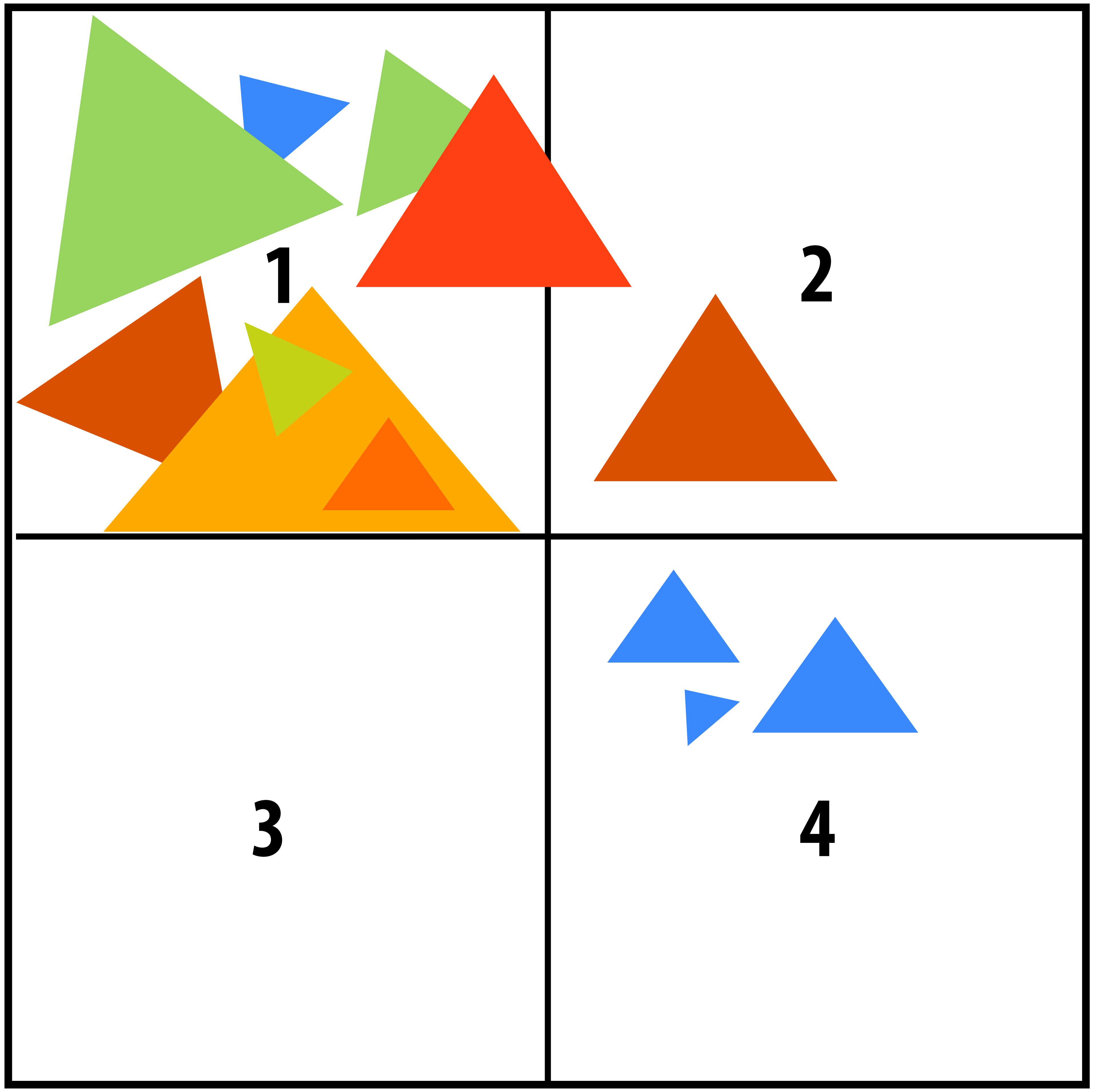


Assign each replicated pipeline responsibility for a region of the output image

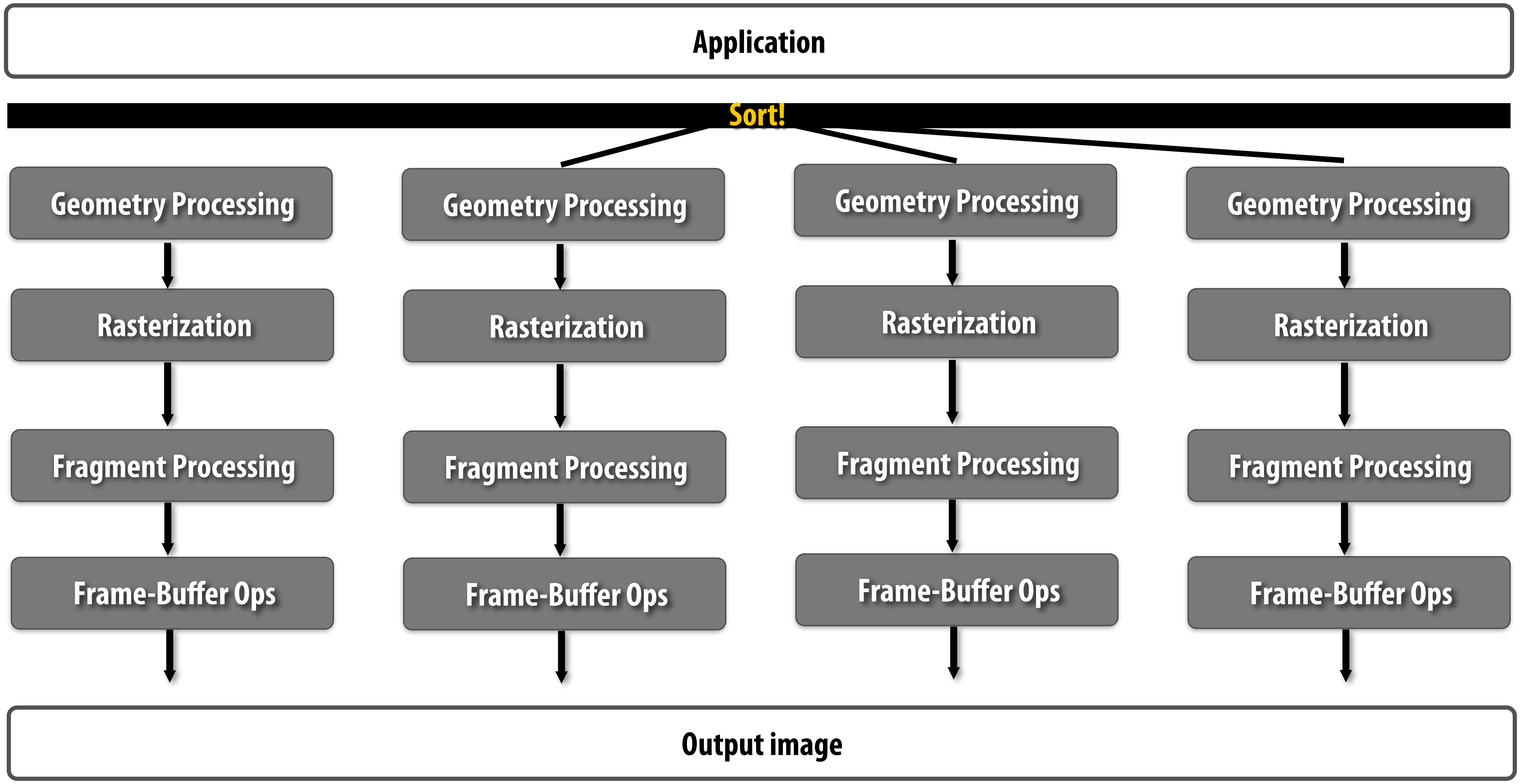
Do minimal amount of work (compute screen-space vertex positions of triangle) to determine which region(s) each input primitive overlaps

Sort first work partitioning

(partition the primitives to parallel units based on screen overlap)

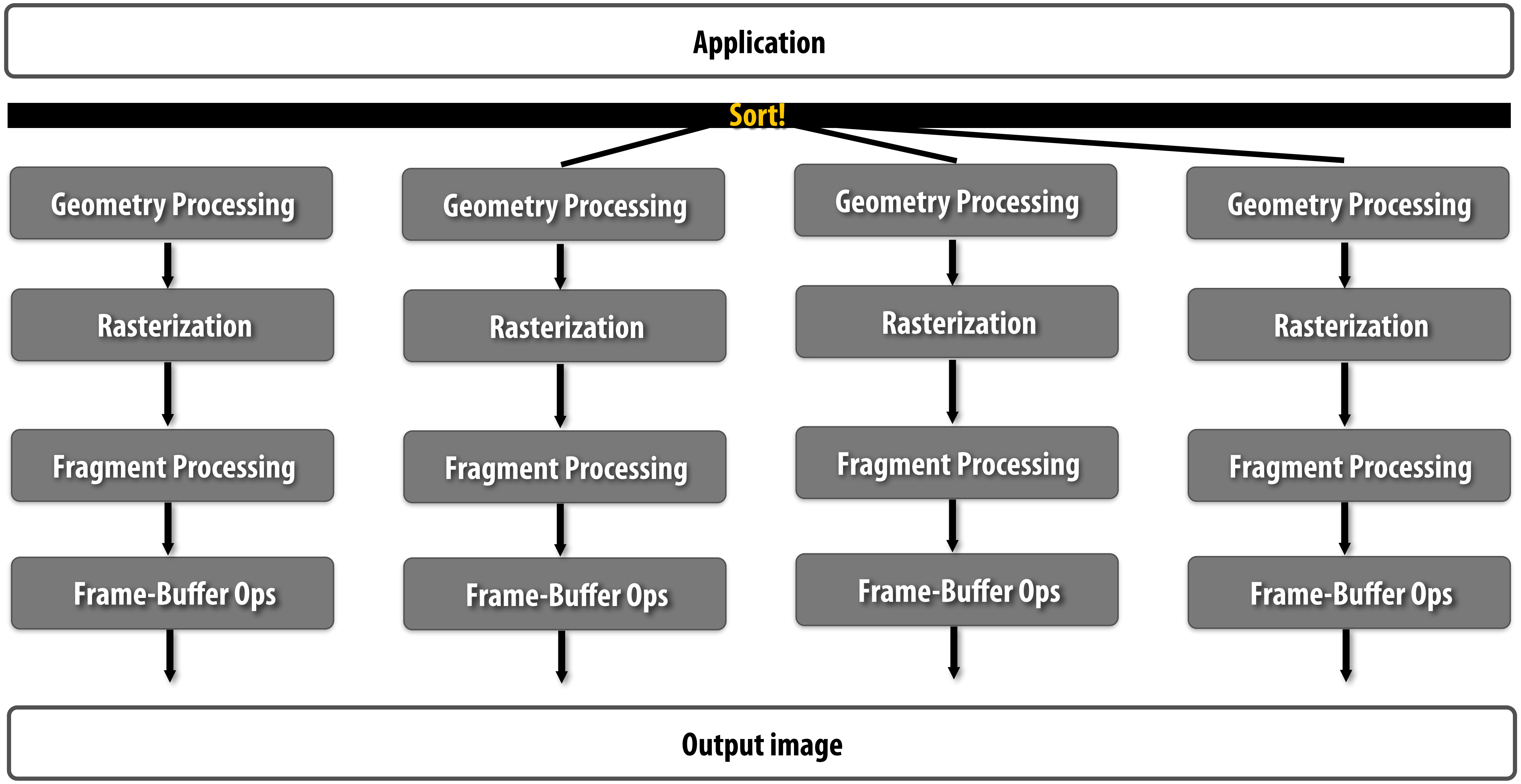


Sort first



- Good:
  - Simple parallelization: just replicate rendering pipeline and operate independently (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

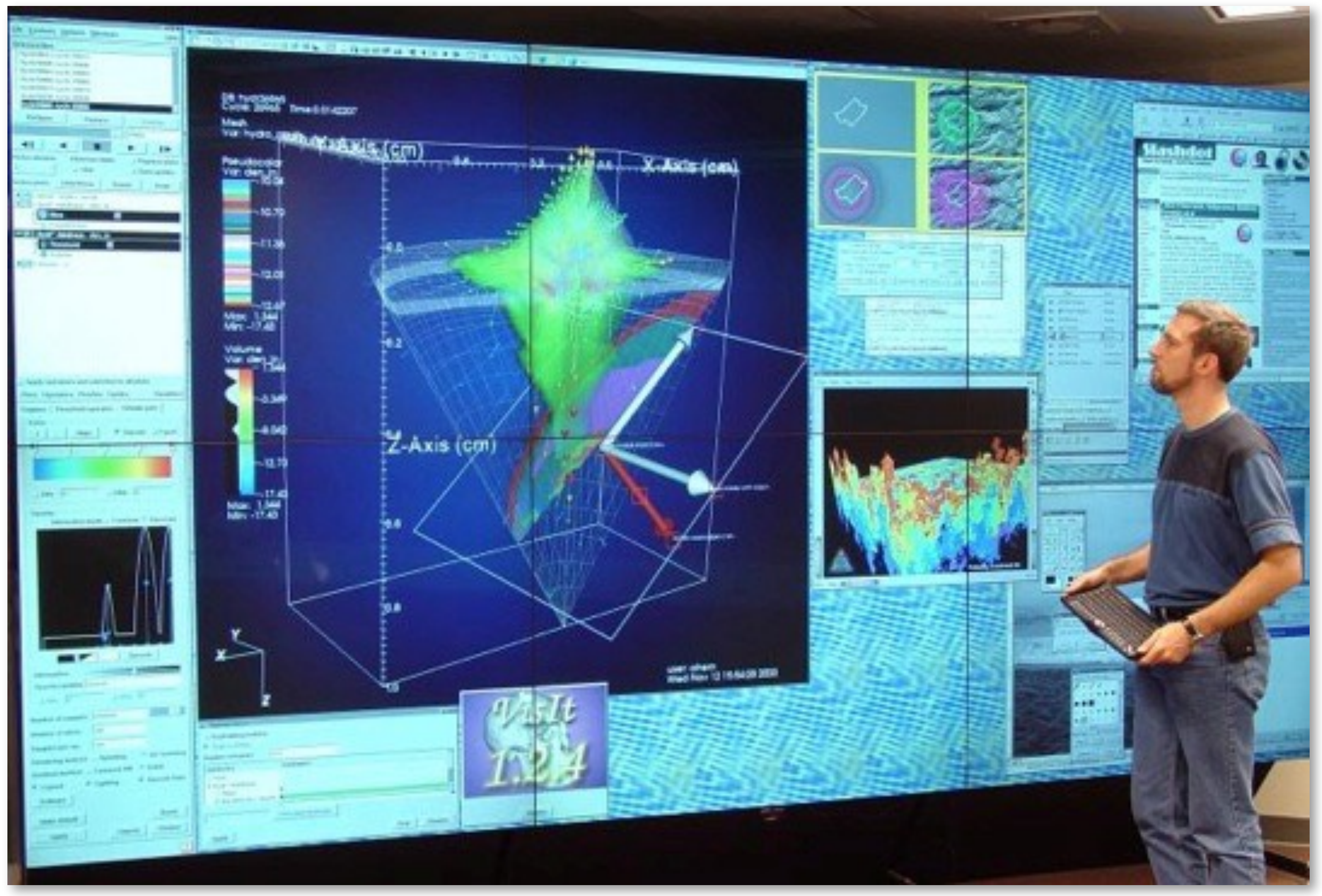
Sort first



- **Bad:**
  - **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)

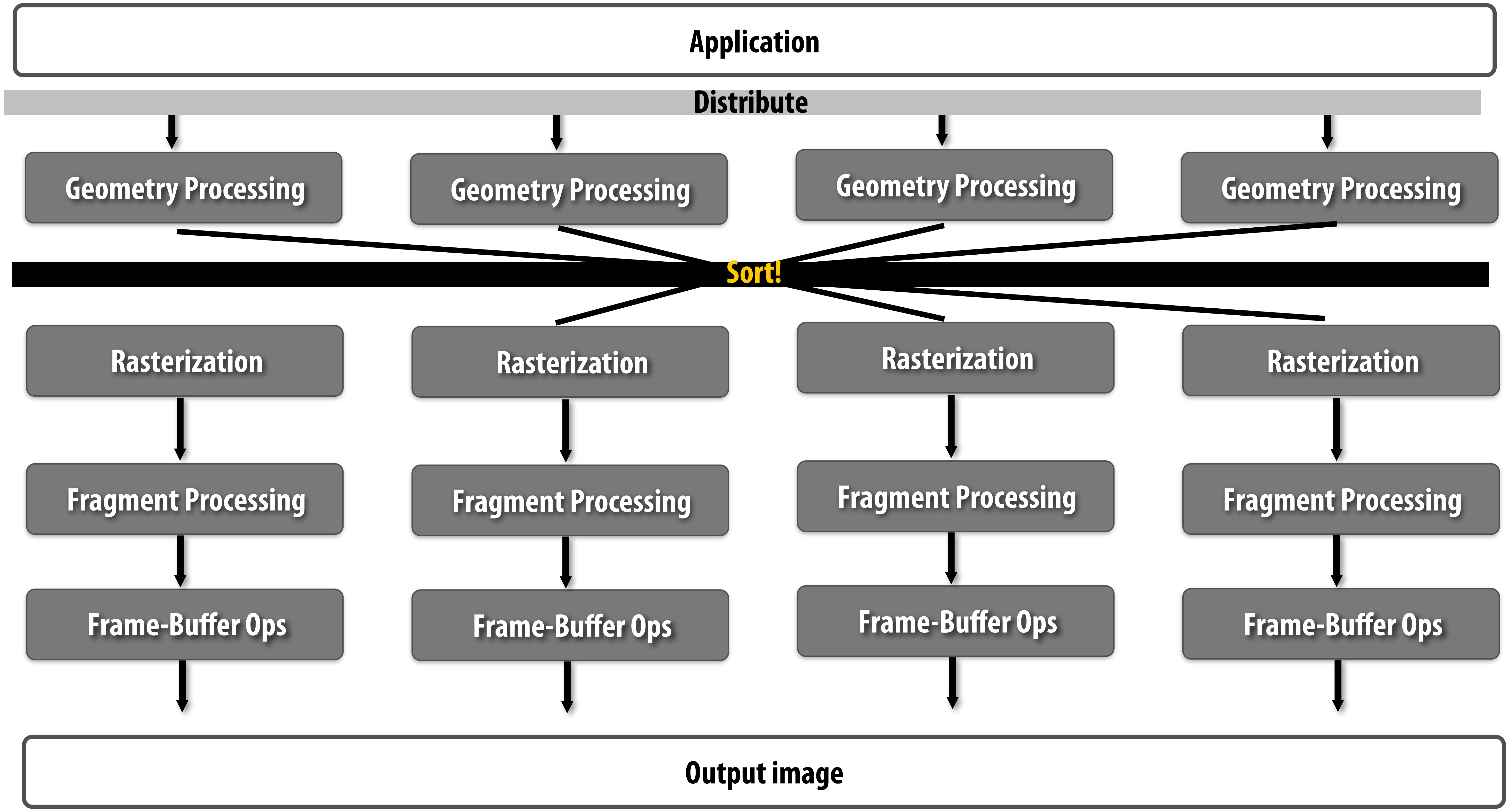


- Pixar's RenderMan
  - Multi-core software renderer
  - Sort surfaces into screen tiles prior to tessellation

* Chromium can also be configured as a sort-last image composition system

Sort middle

Sort middle



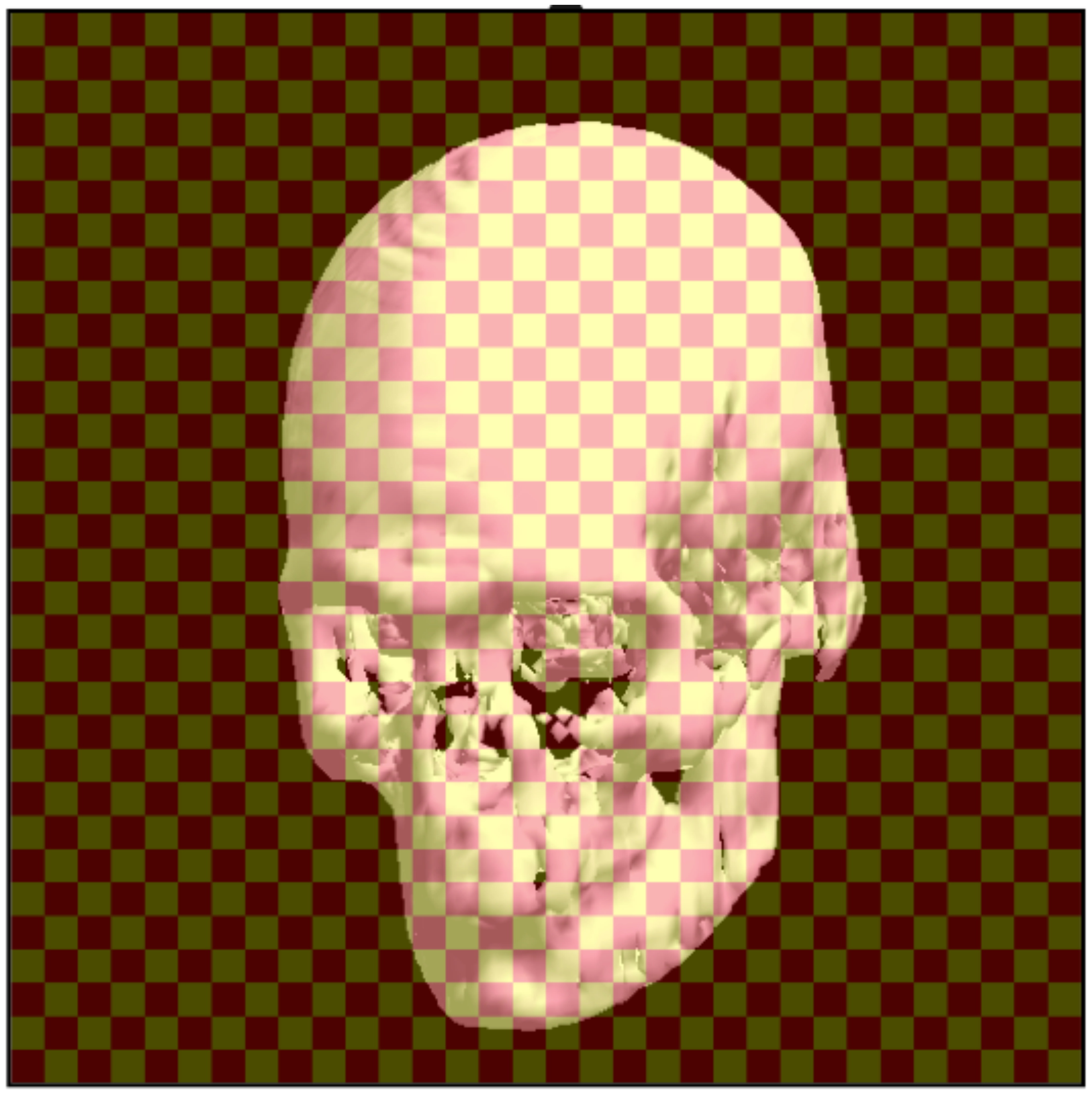
Distribute primitives to pipelines (e.g., round-robin distribution)

Assign each rasterizer a region of the render target

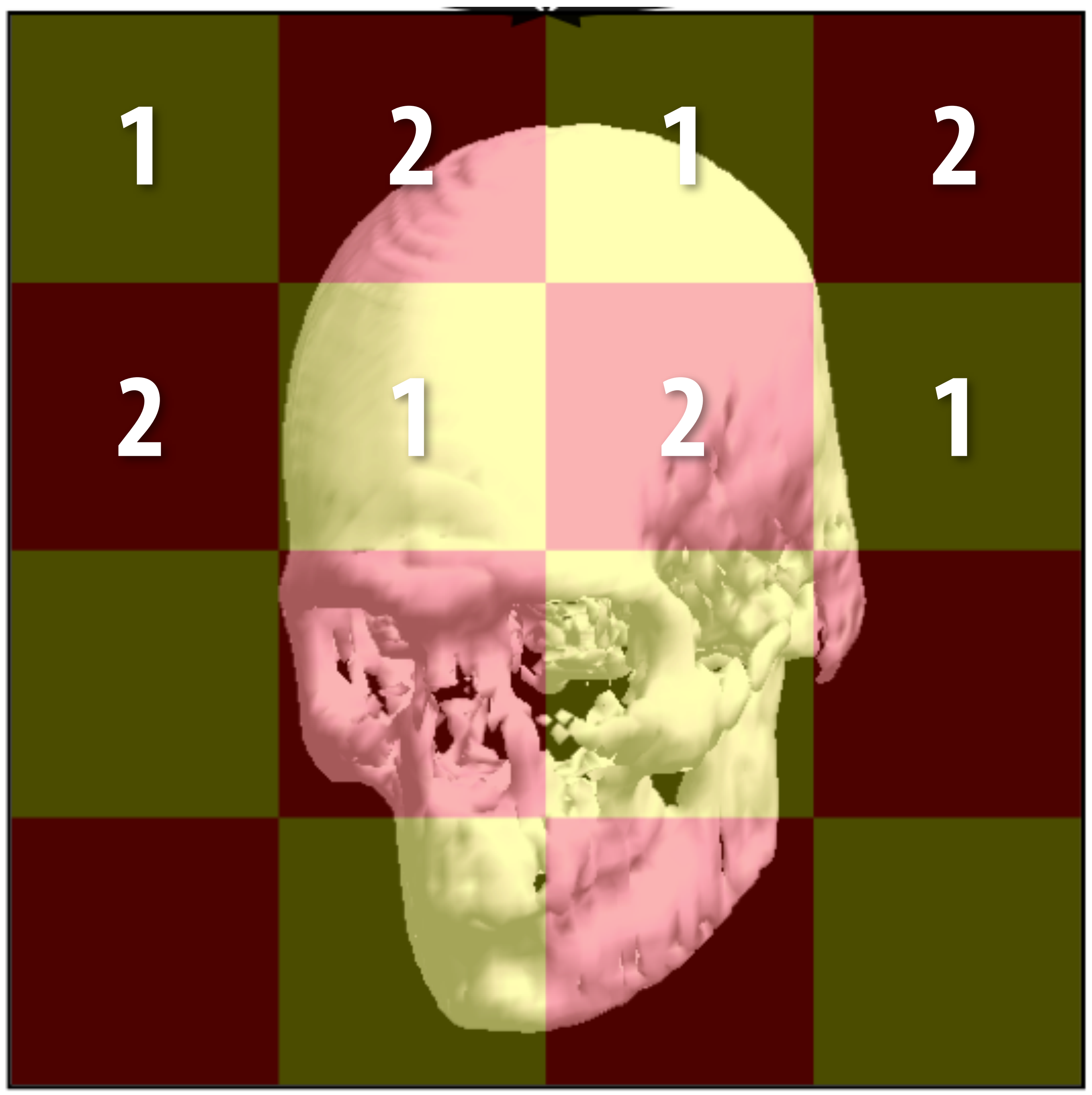
Sort after geometry processing based on screen space projection of primitive vertices

Interleaved mapping of screen

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



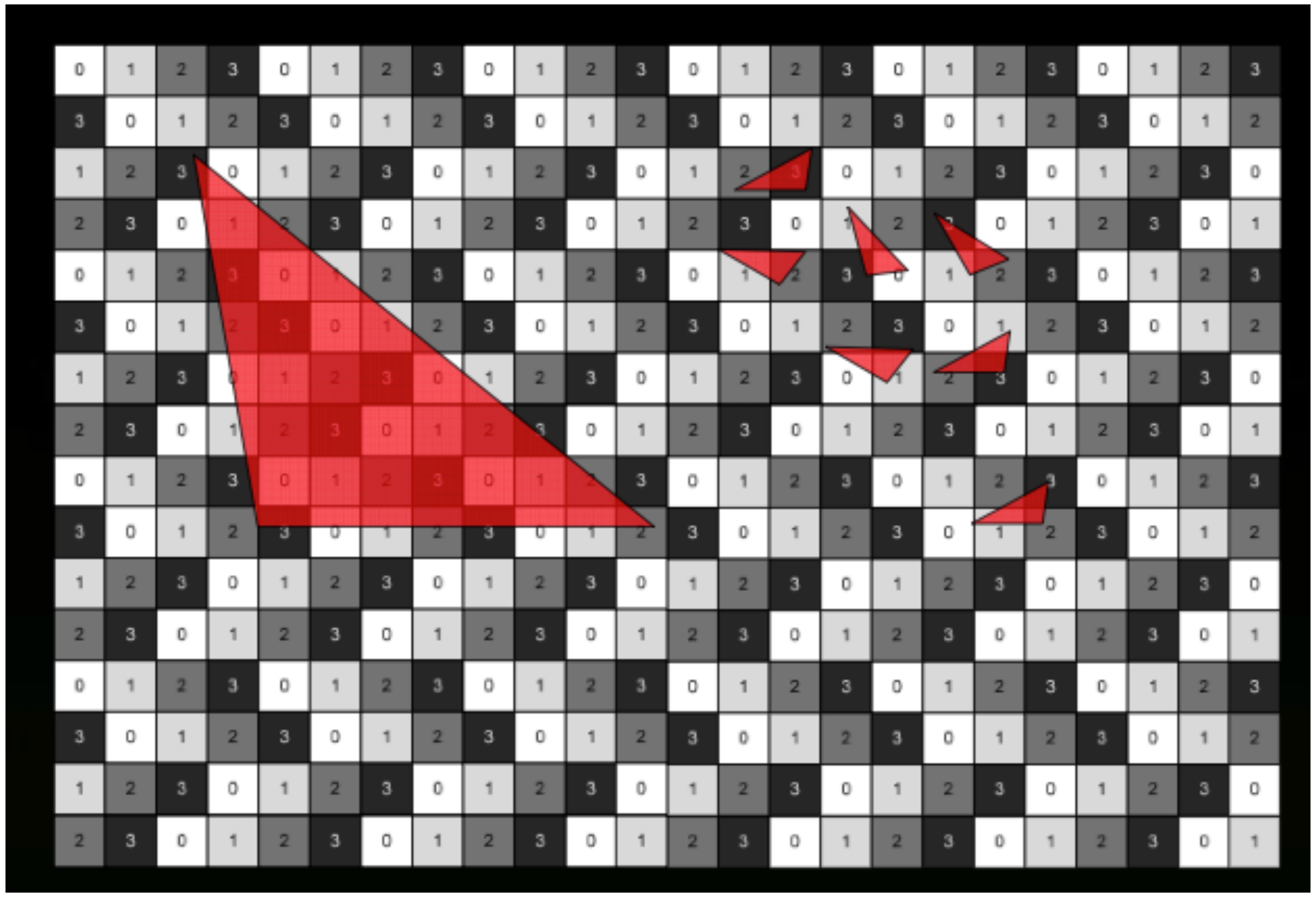
Interleaved mapping



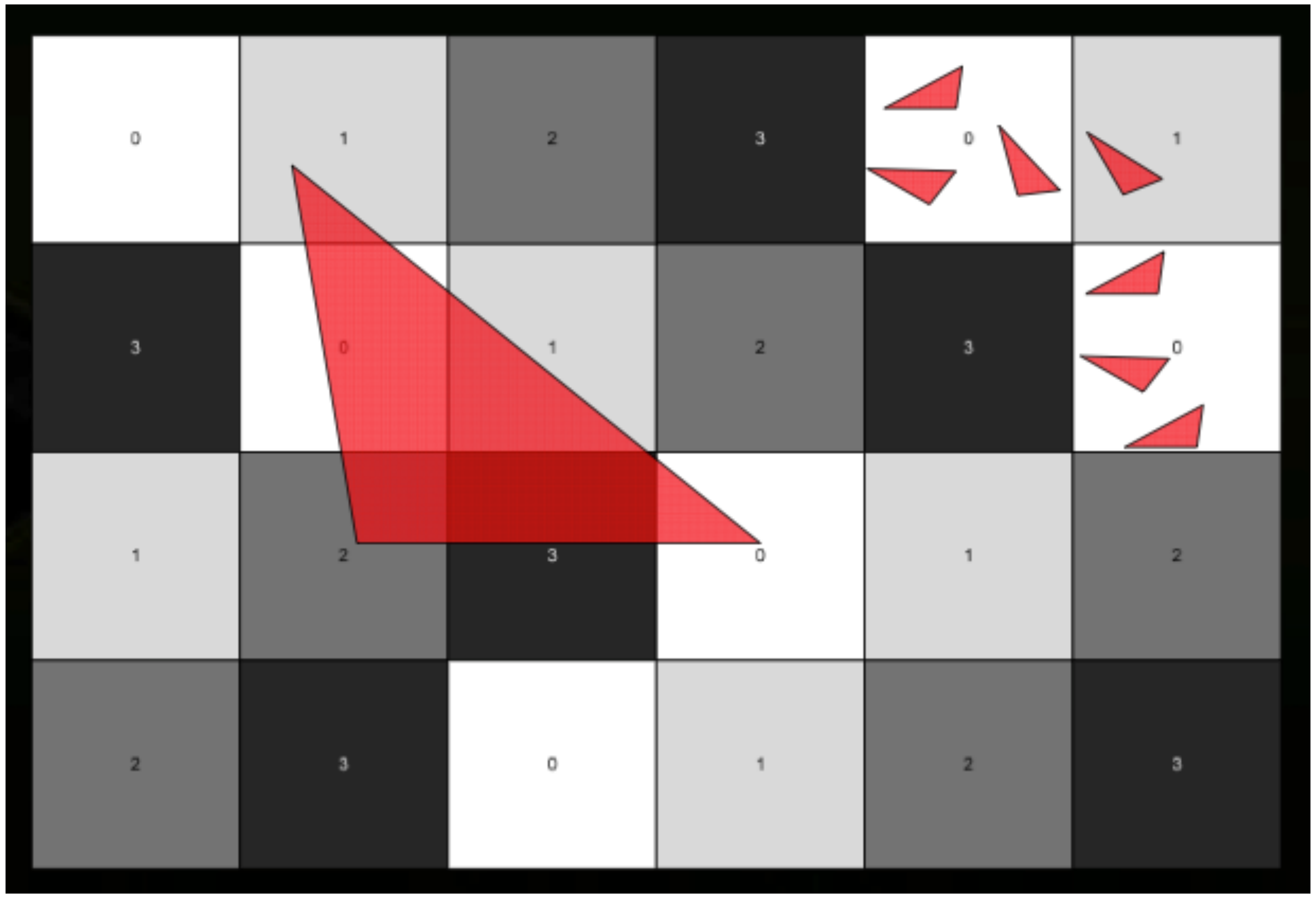
Tiled mapping

Fragment interleaving in NVIDIA Fermi

Fine granularity interleaving



Coarse granularity interleaving

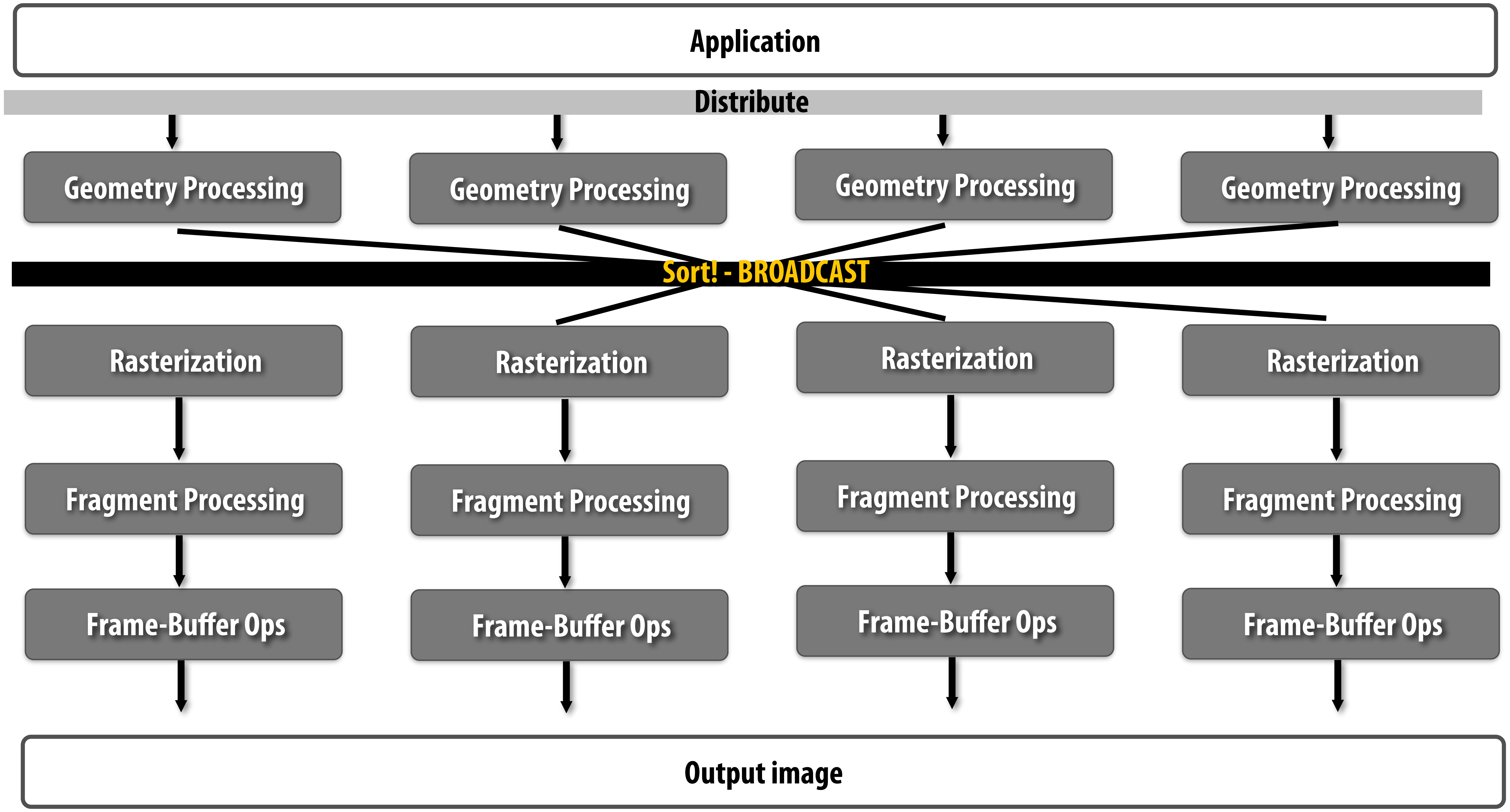


Question 1: what are the benefits/weaknesses of each interleaving?

Question 2: notice anything interesting about these patterns?

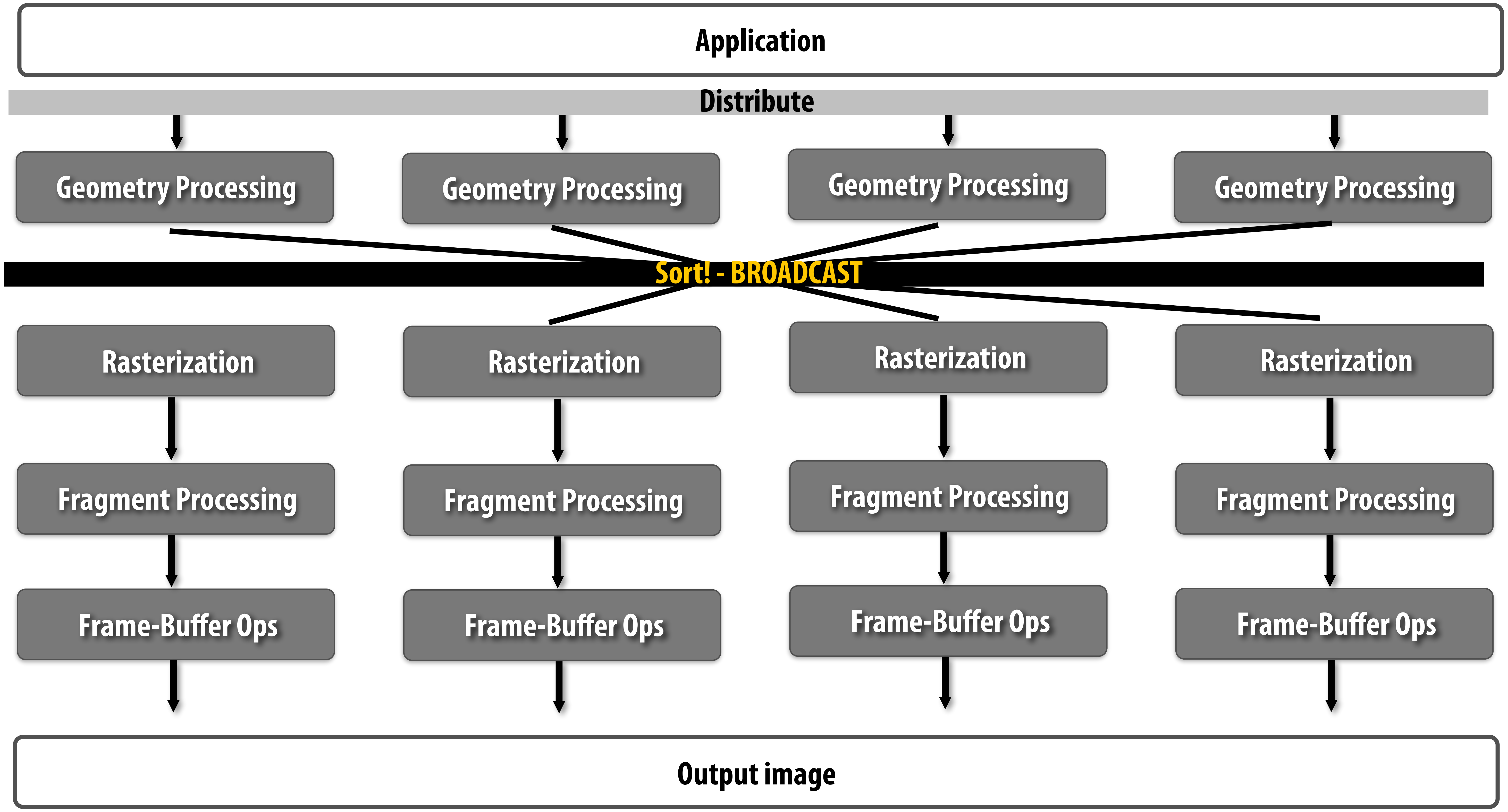
[Image source: NVIDIA]

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

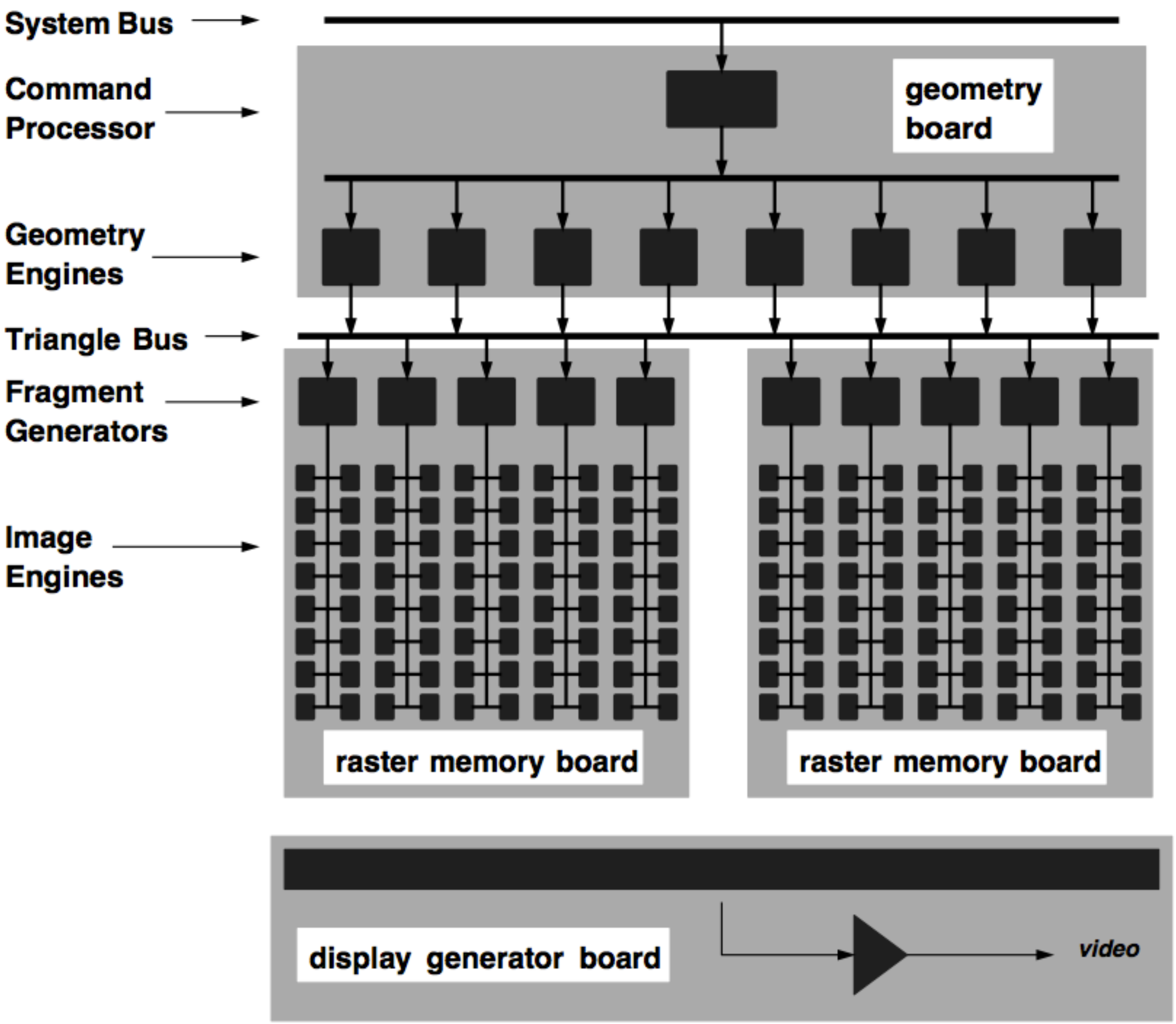
Sort middle interleaved



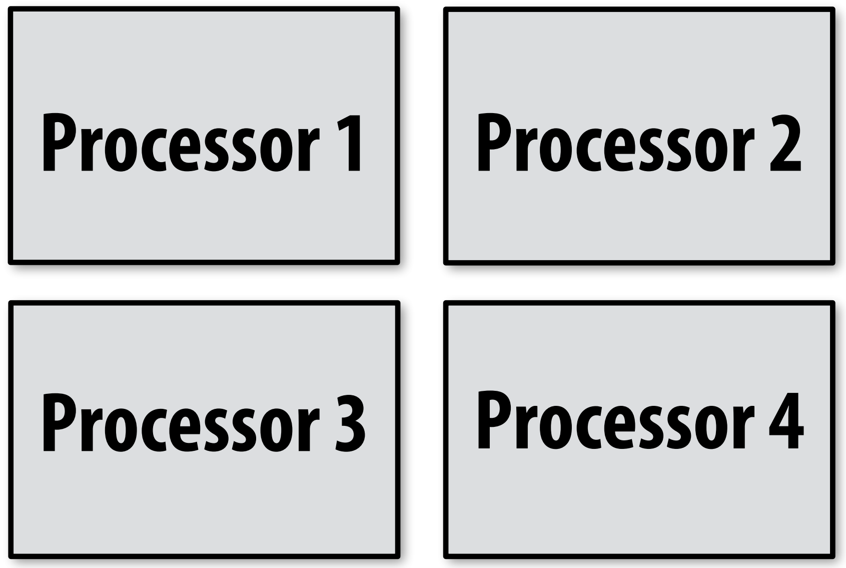
- Bad:
  - Bandwidth scaling: sort is implemented as a broadcast
    (each triangle goes to many/all rasterizers because of interleaved screen mapping)
  - If tessellation is enabled, must communicate many more primitives than sort first
  - Duplicated per triangle setup work across rasterizers

SGI RealityEngine [Akeley 93]

Sort-middle interleaved design



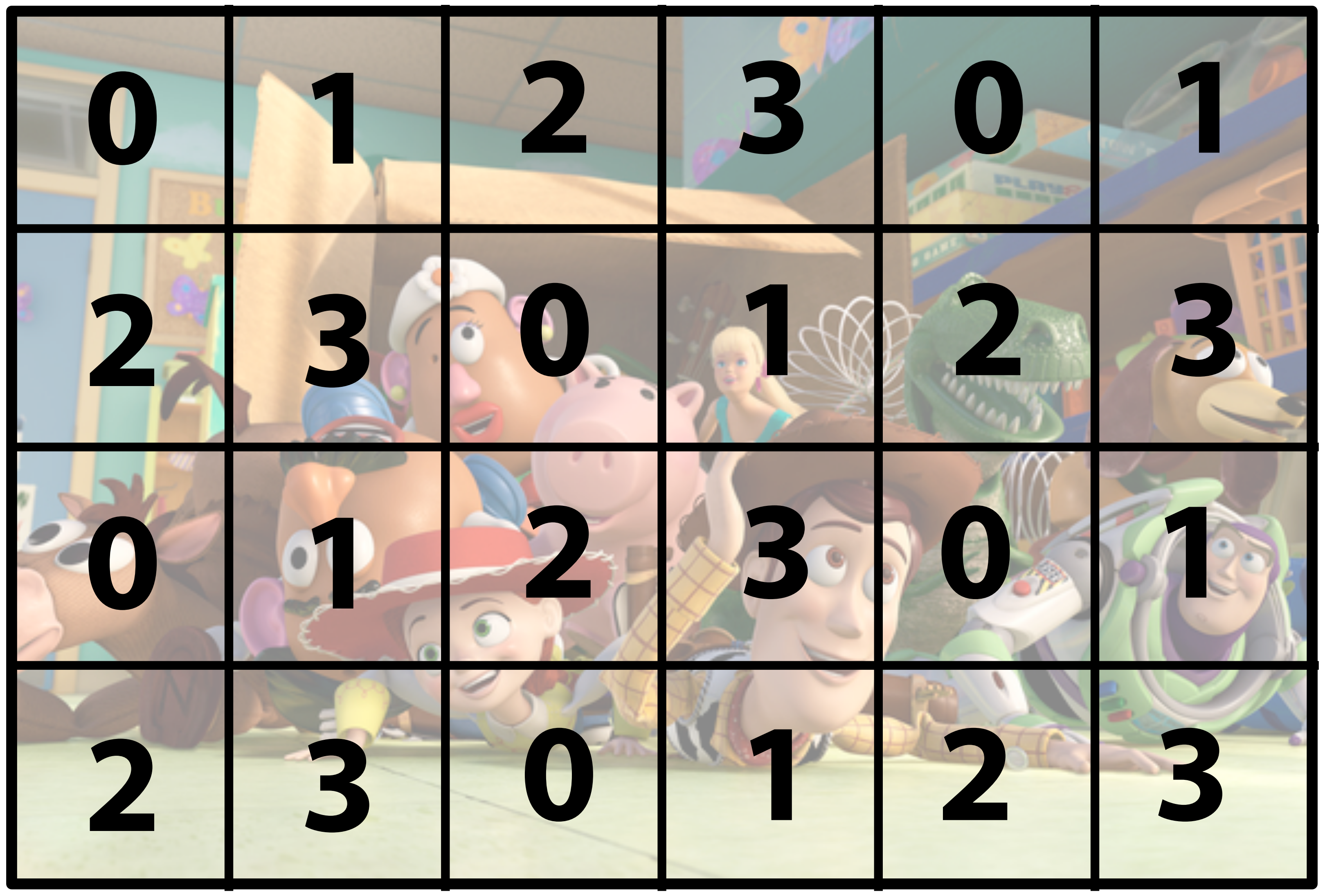
Tiling (a.k.a. "chunking", "bucketing")


Processor 1

Processor 2

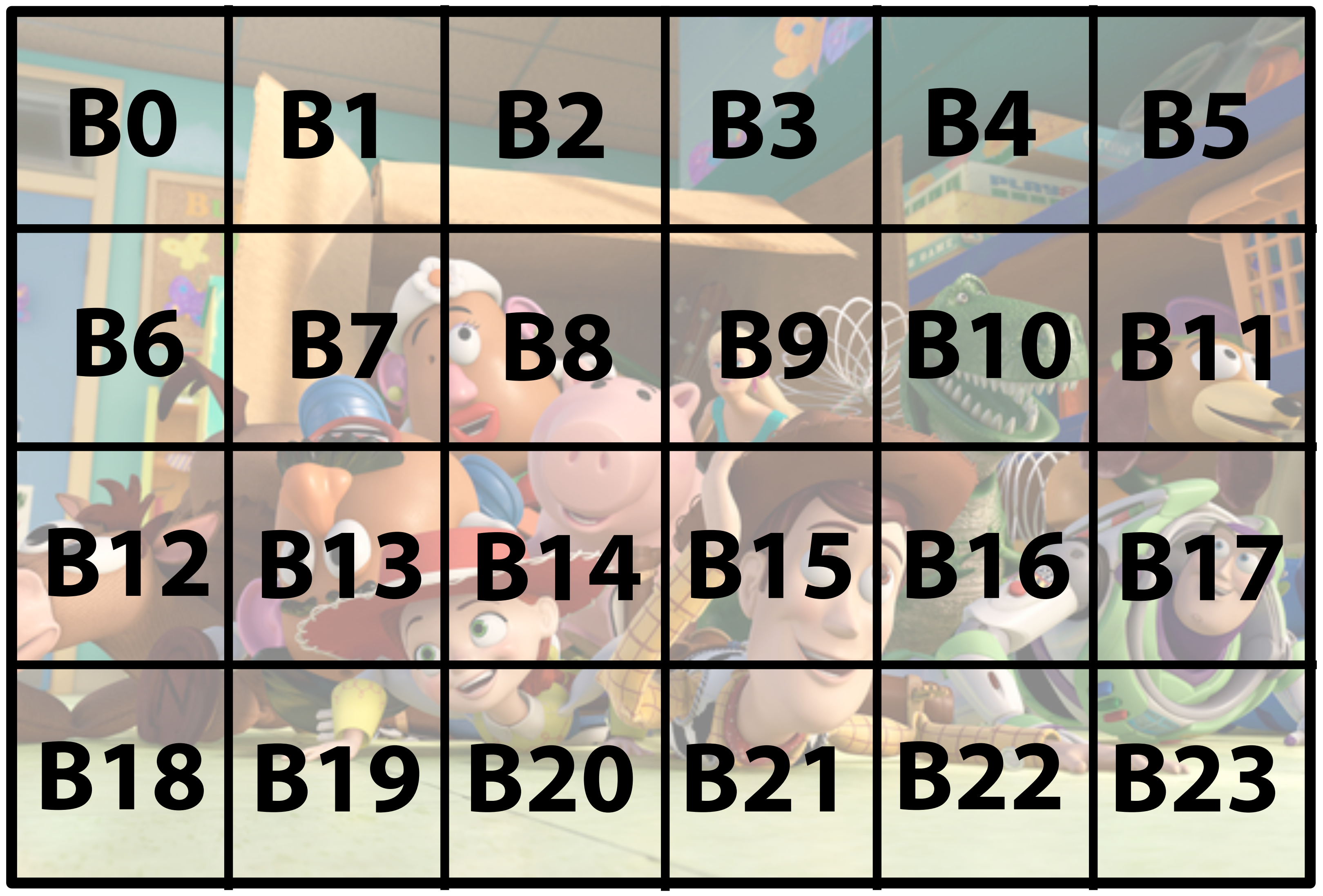
Processor 3

Processor 4



0	1	2	3	0	1
2	3	0	1	2	3
0	1	2	3	0	1
2	3	0	1	2	3

Interleaved (static) assignment of screen tiles to processors

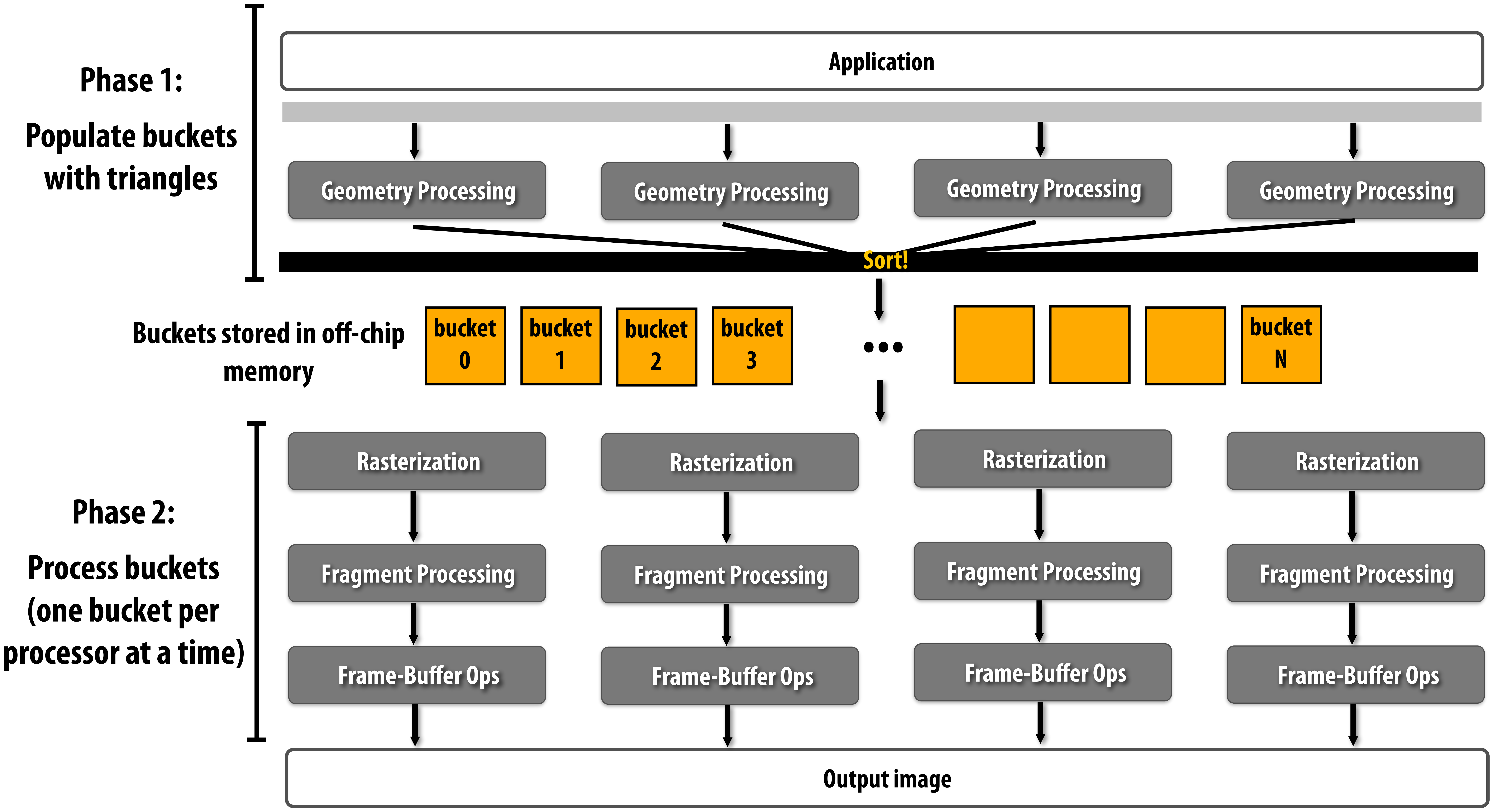


B0	B1	B2	B3	B4	B5
B6	B7	B8	B9	B10	B11
B12	B13	B14	B15	B16	B17
B18	B19	B20	B21	B22	B23

Assignment to buckets

List of buckets is a work queue. Buckets are dynamically assigned to processors.

Sort middle tiled (chunked)



Partition screen into many small tiles (many more tiles than physical rasterizers)

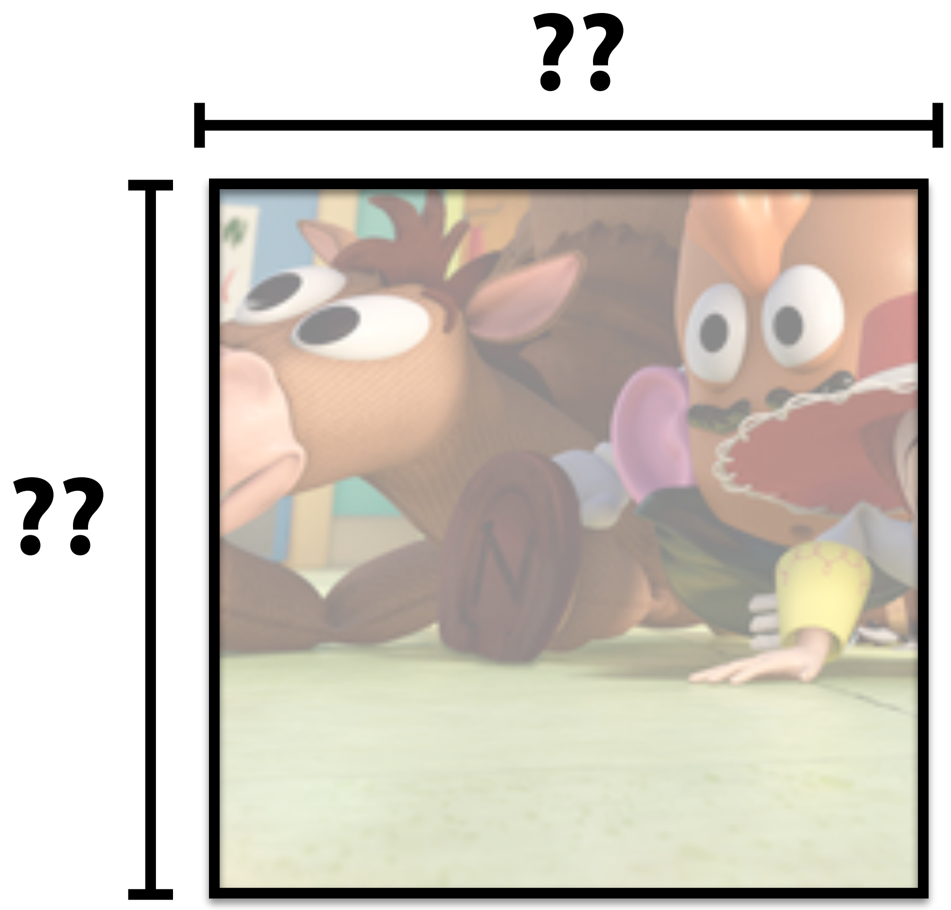
Sort geometry by tile into buckets (one bucket per tile of screen)

After all geometry is bucketed, rasterizers process buckets in parallel

Sort middle tiled (chunked)

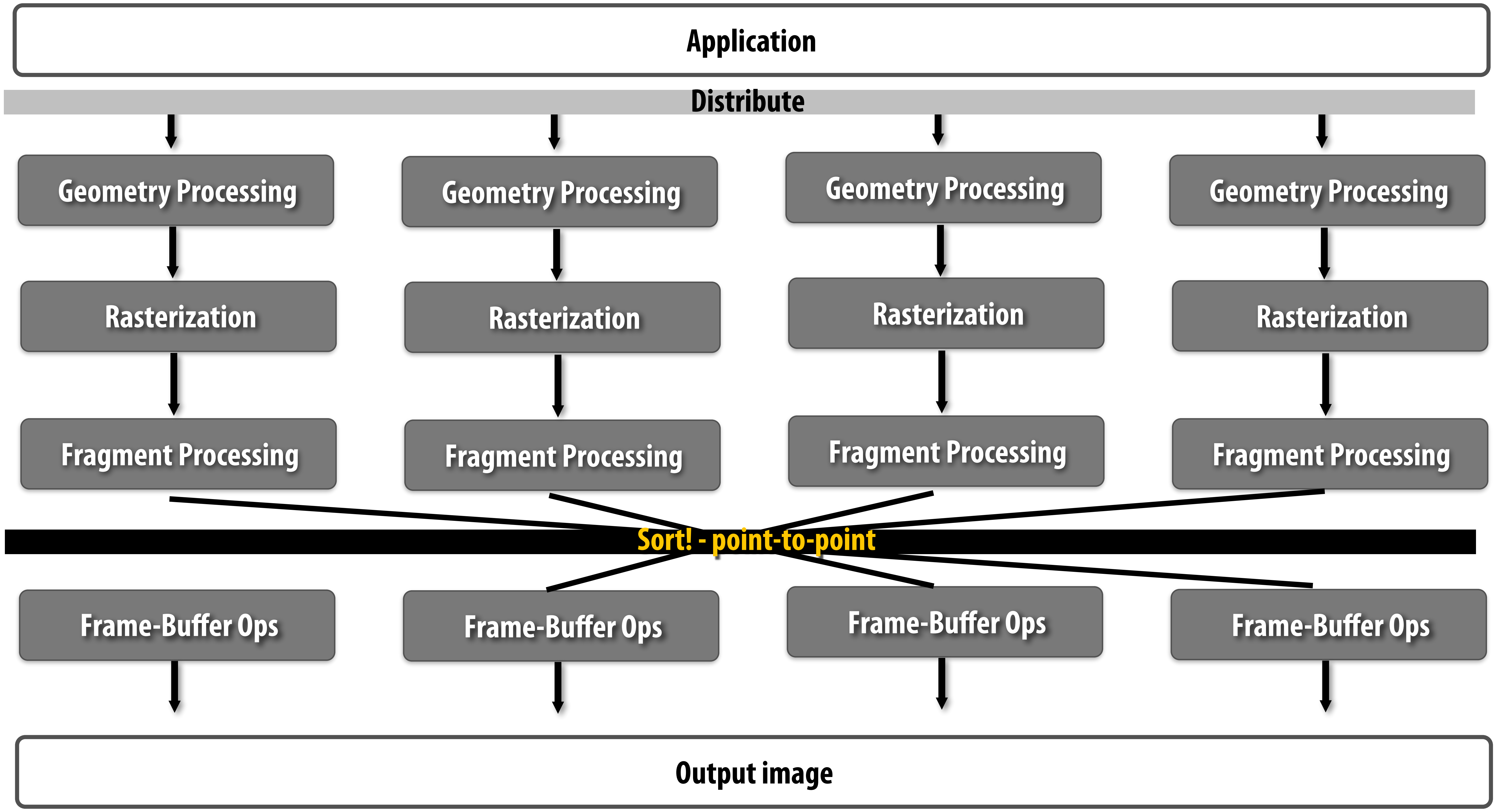
- Good:
  - Good load balance (distribute many buckets onto rasterizers)
  - Potentially low bandwidth requirements (why? when?)
    - Question: What should the size of tiles be for maximum BW savings?
  - Challenge: "bucketing" sort has low contention (assuming each triangle only touches a small number 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

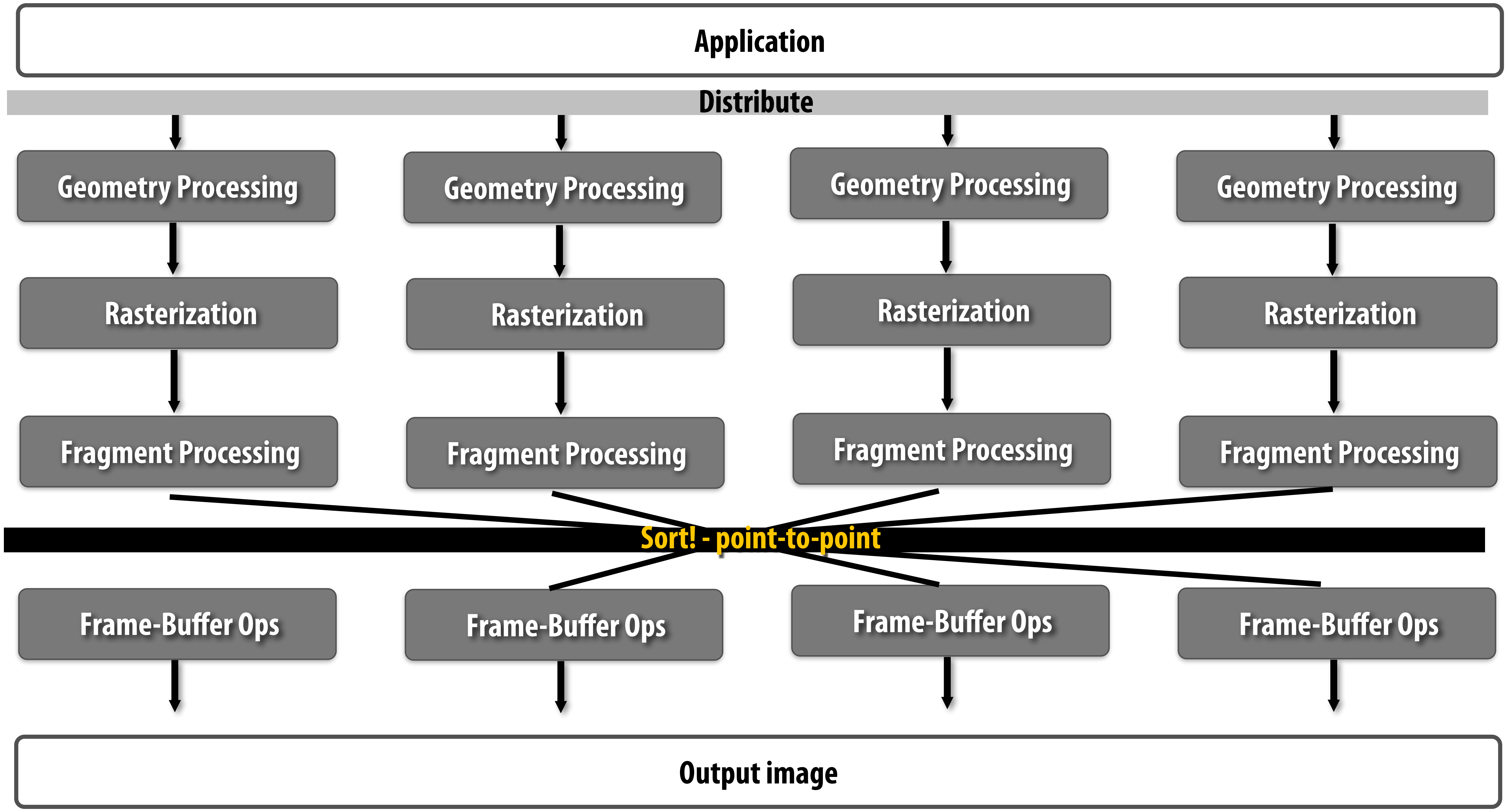
Sort last fragment



Distribute primitives to top of pipelines (e.g., round robin)

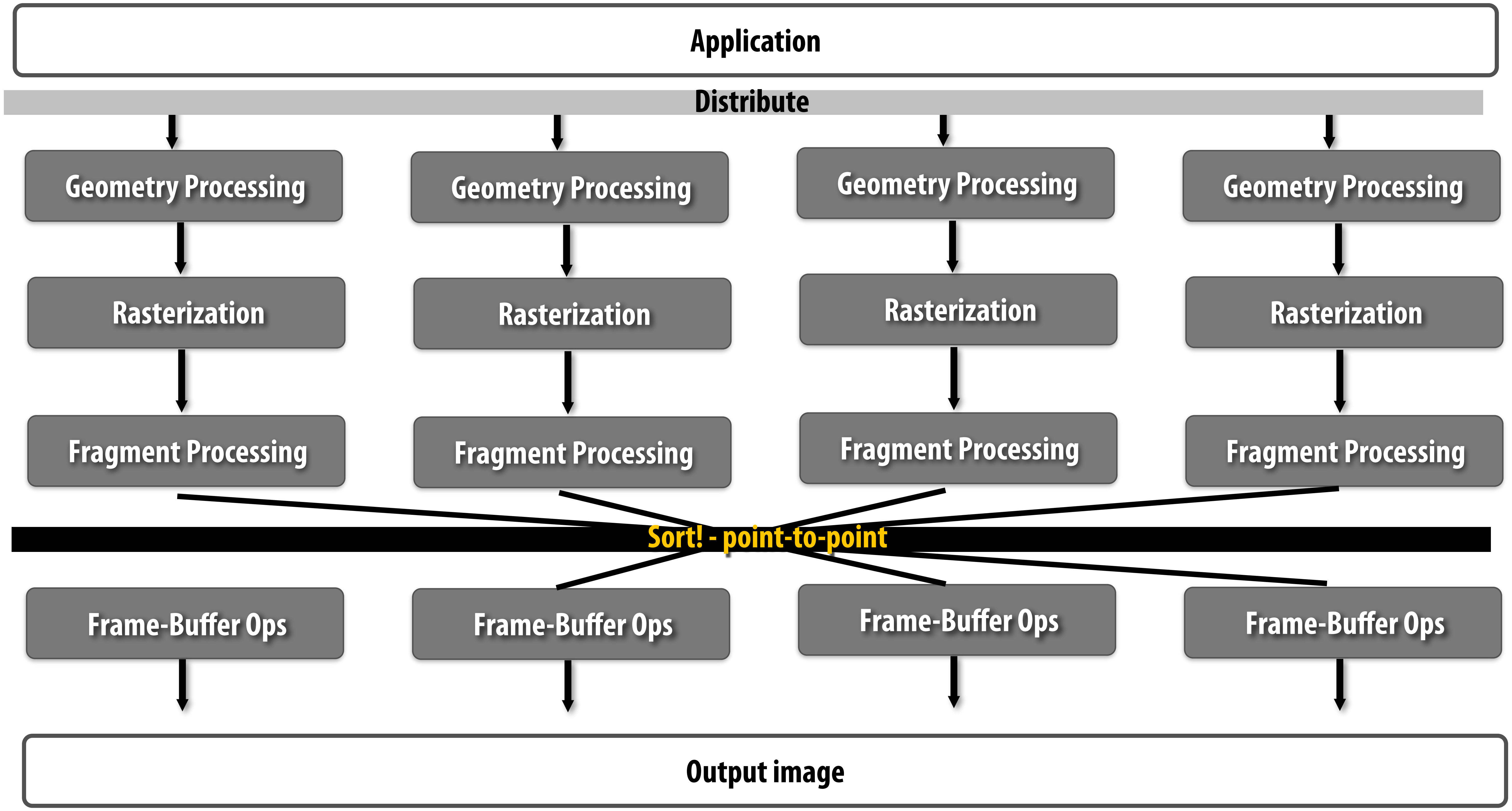
Sort after fragment processing based on (x,y) position of fragment

Sort last fragment



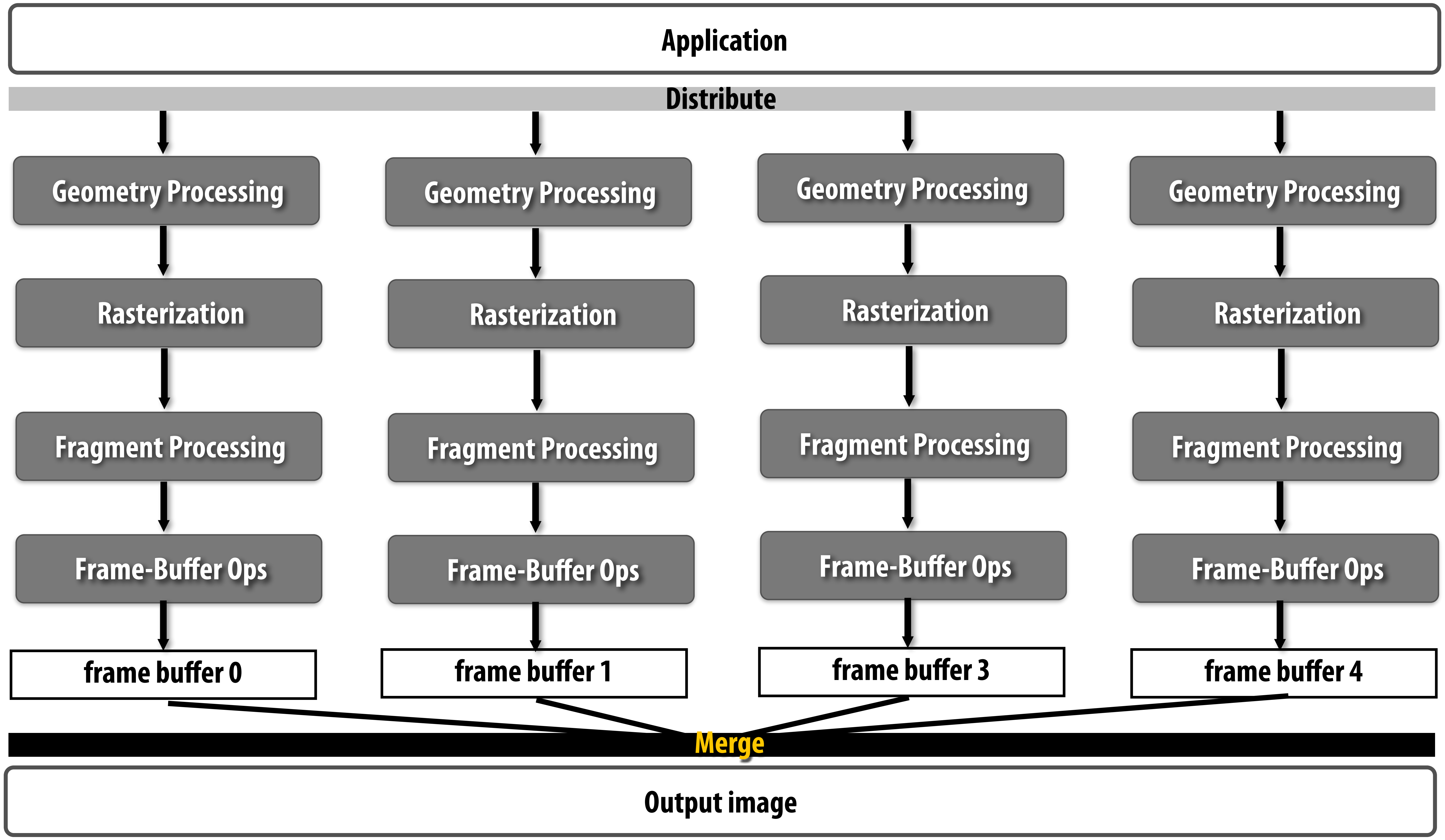
- **Good:**
  - **No redundant geometry processing or rasterizeration (but early z-cull is a problem)**
  - **Point-to-point communication during sort**
  - **Interleaved pixel mapping results in good workload balance for frame-buffer ops**

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)**

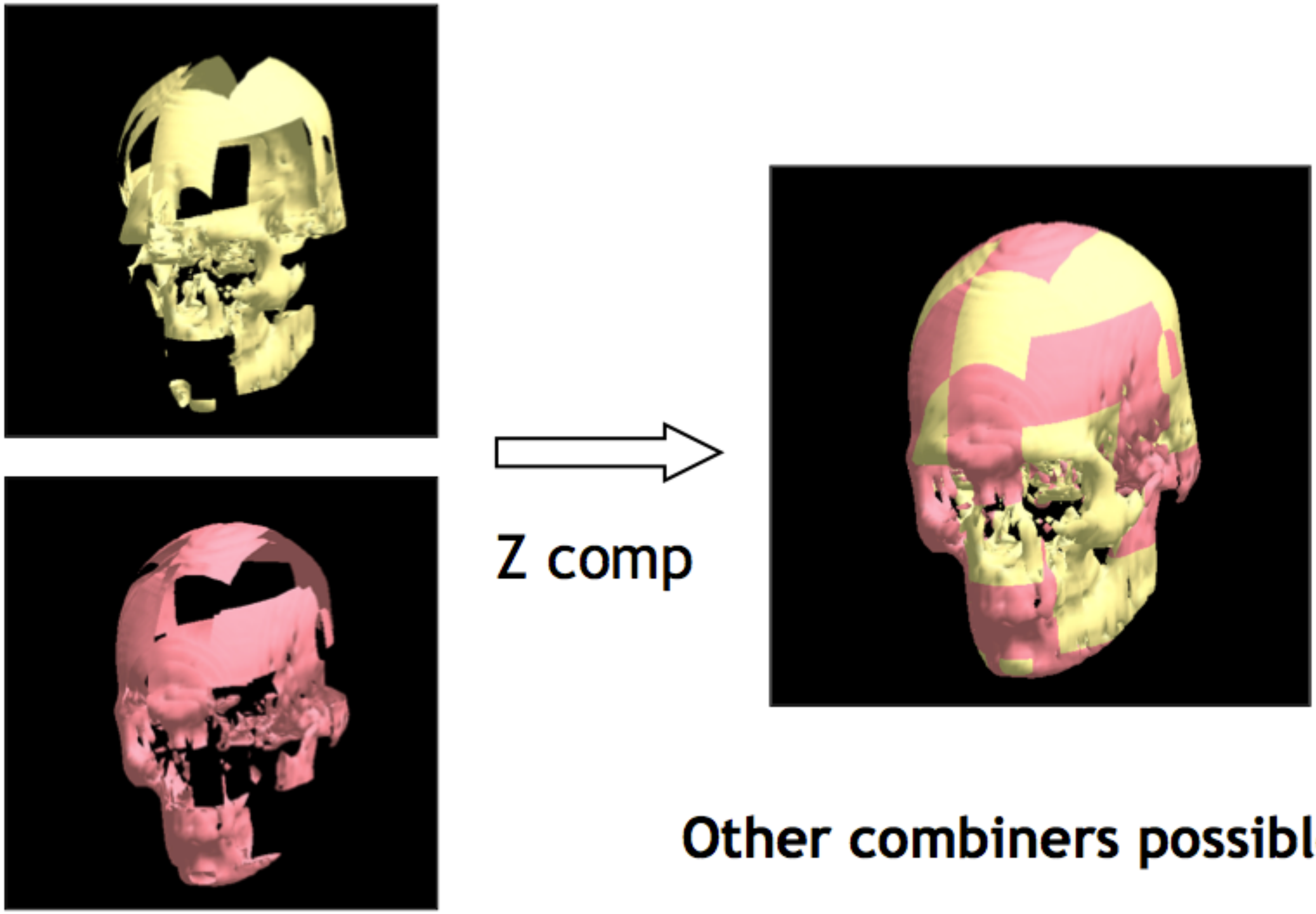
Sort last image composition



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

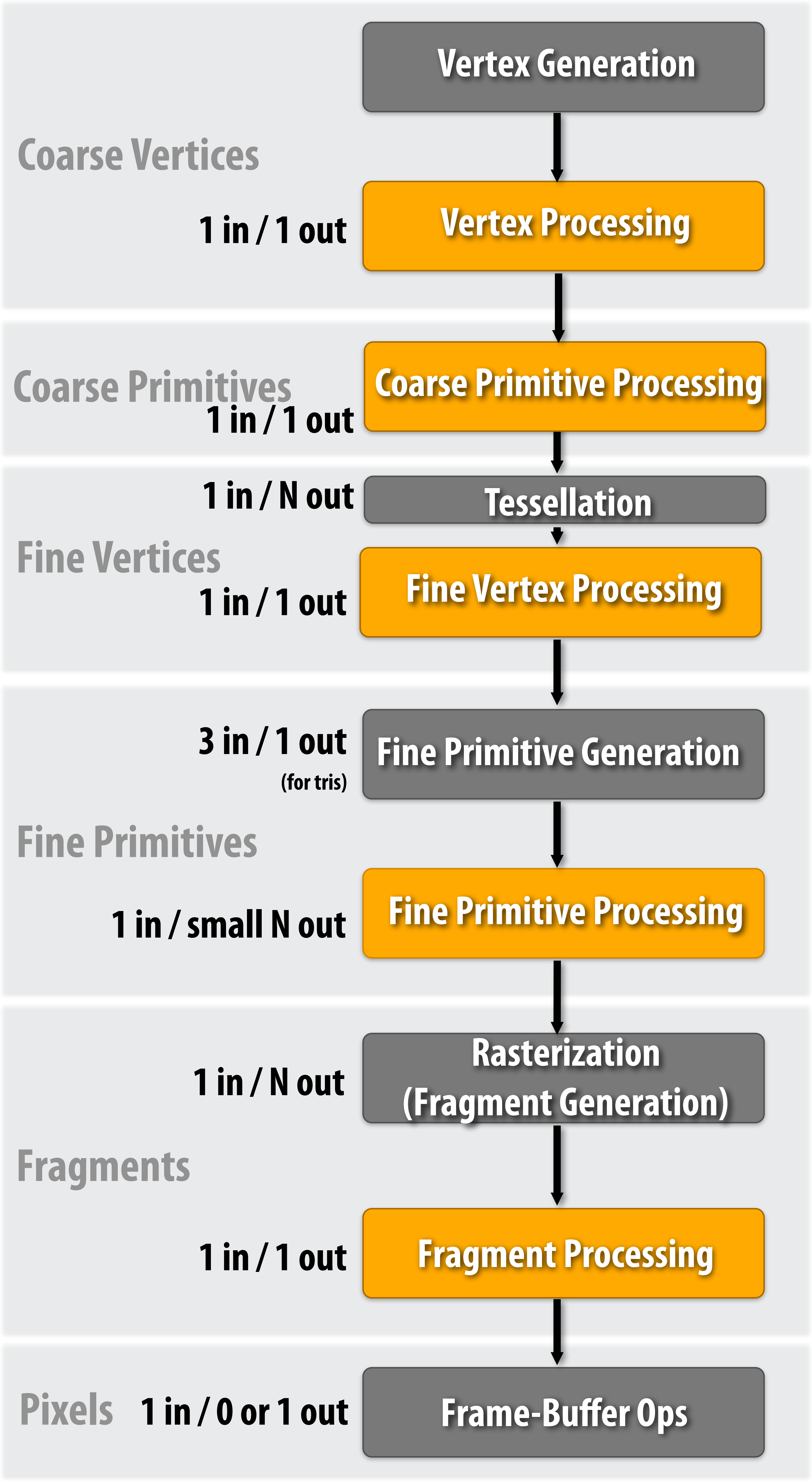
- Simple implementation: N separate rendering pipelines
  - Can use off-the-shelf GPUs to build a massive rendering system
  - Coarse-grained communication (image buffers)

- Similar load imbalance problems as sort-last fragment

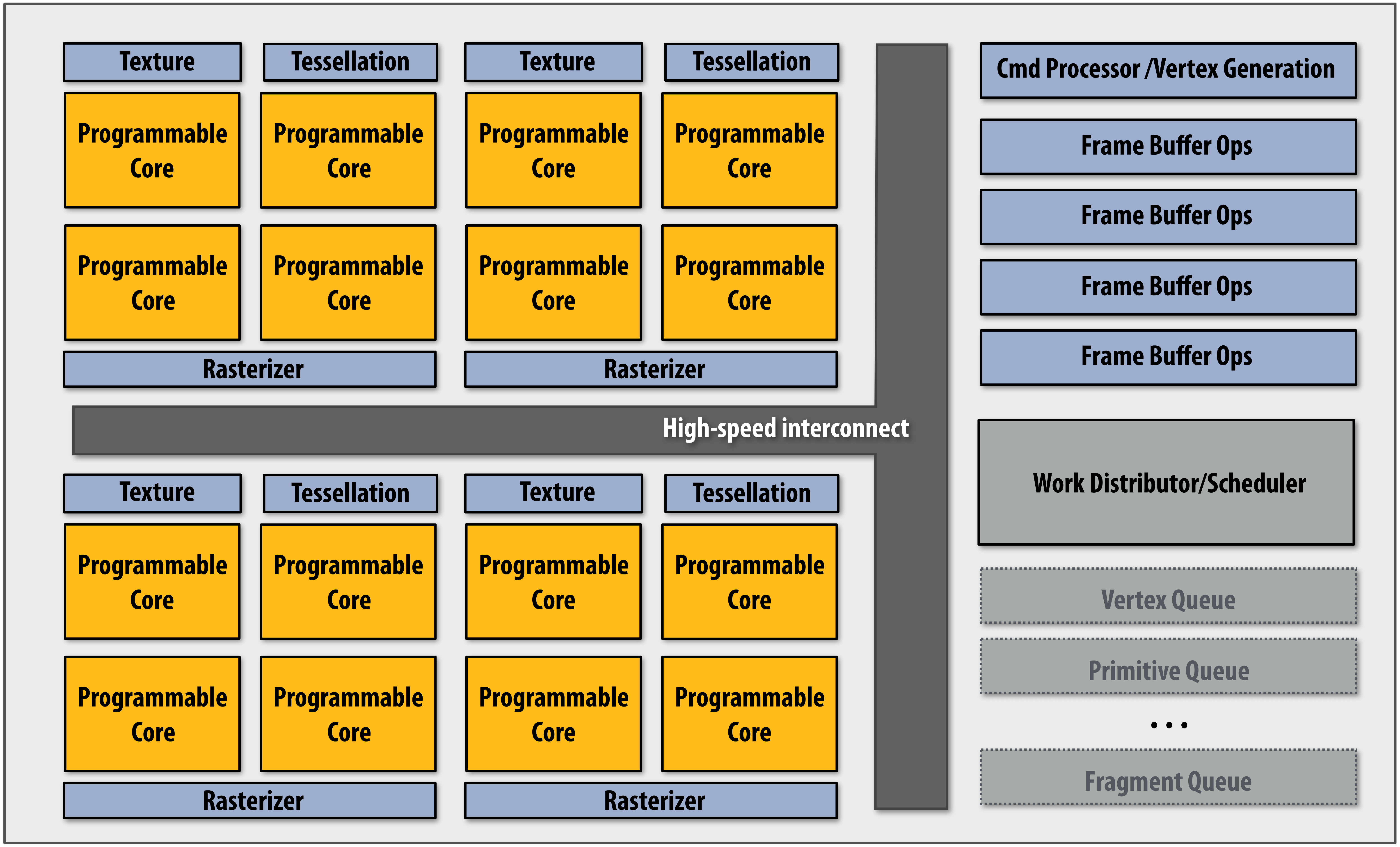
- Under high depth complexity, bandwidth requirement is lower than sort last fragment
  - Communicate final pixels, not all fragments

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)

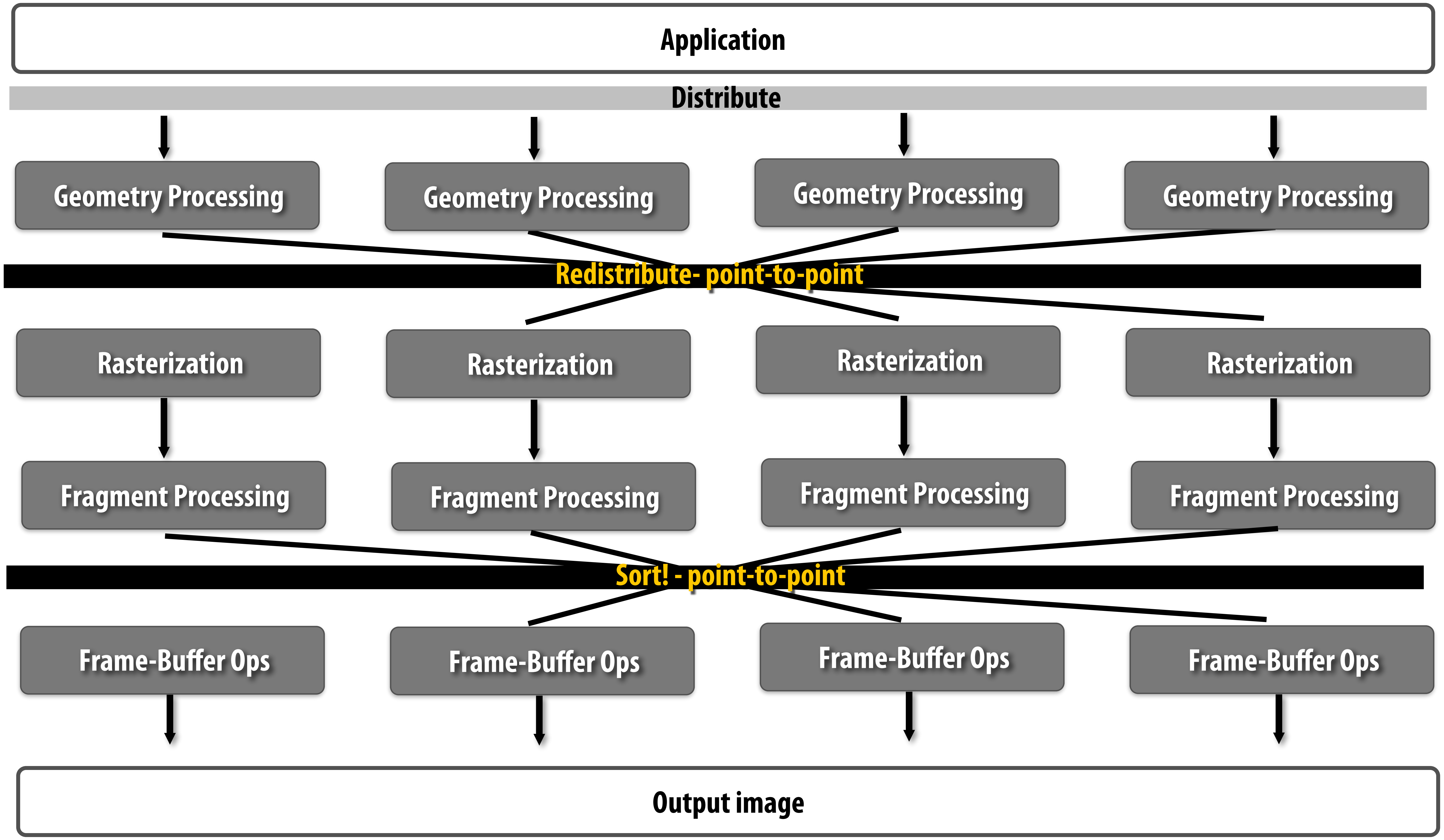
Programmable resources are time-shared by vertex/primitive/fragment processing work
Must keep programmable cores busy: sort everywhere
Hardware work distributor assigns work to cores (based on contents of inter-stage queues)

Sort everywhere

(How modern high-end GPUs are scheduled)

Sort everywhere

[Eldridge 00]



Distribute primitives to top of pipelines

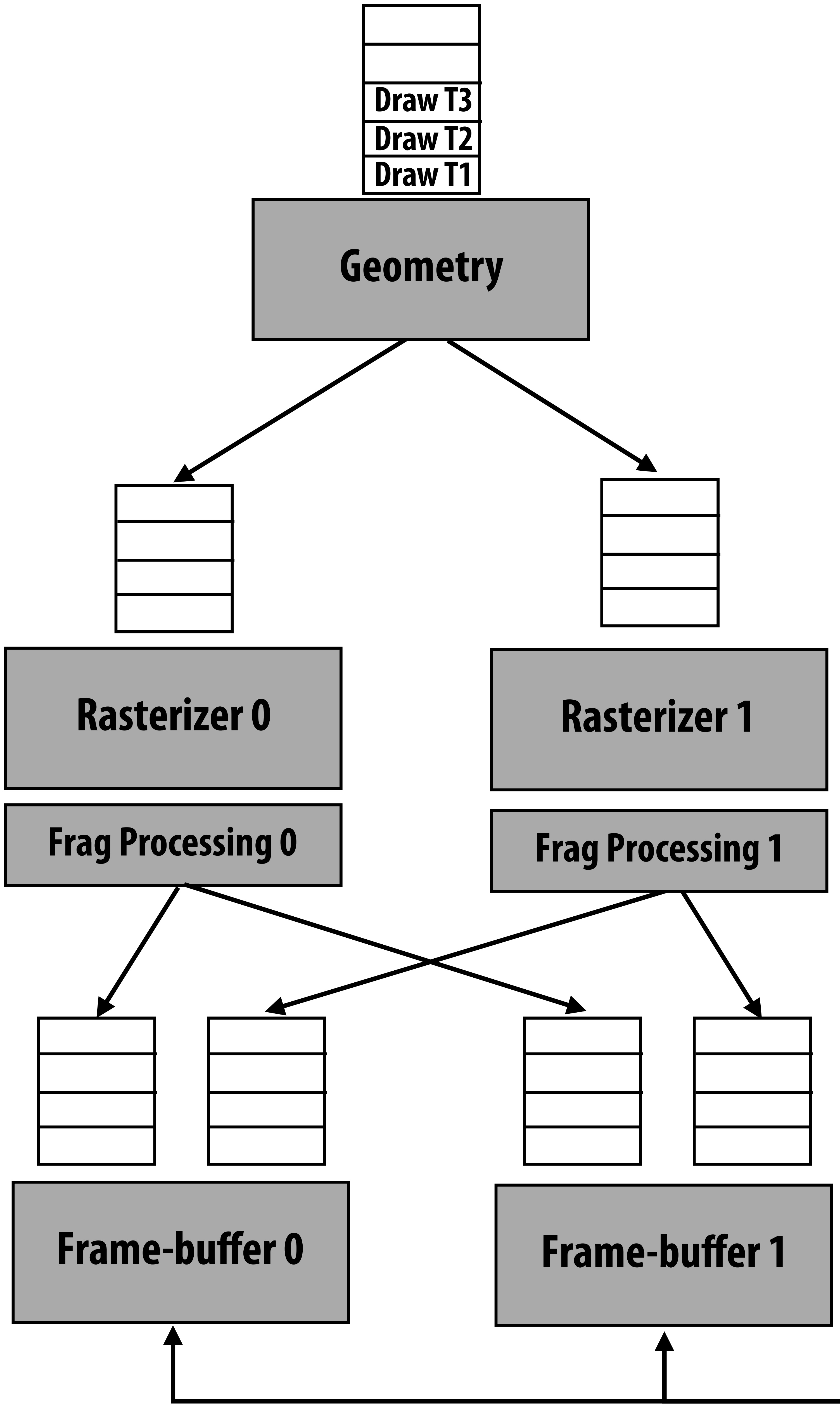
Redistribute after geometry processing (e.g, round robin)

Sort after fragment processing based on (x,y) position of fragment

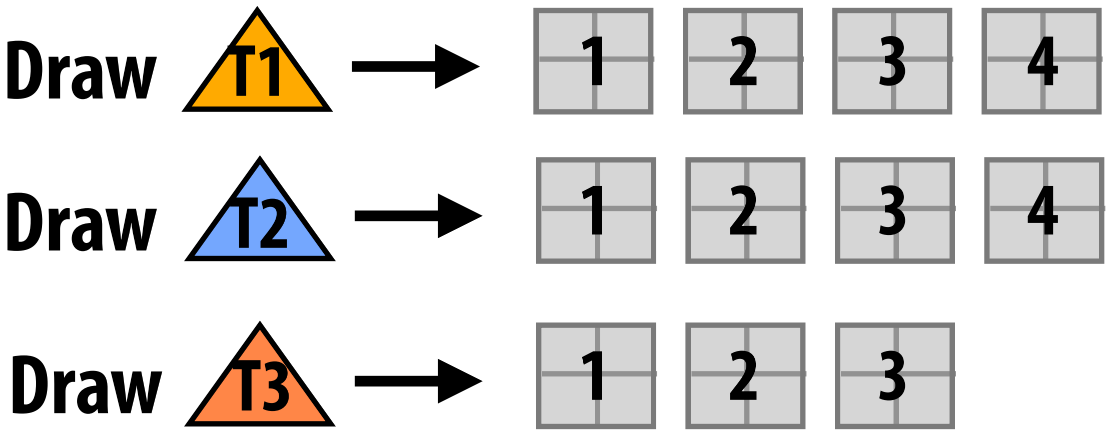
Implementing sort everywhere

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

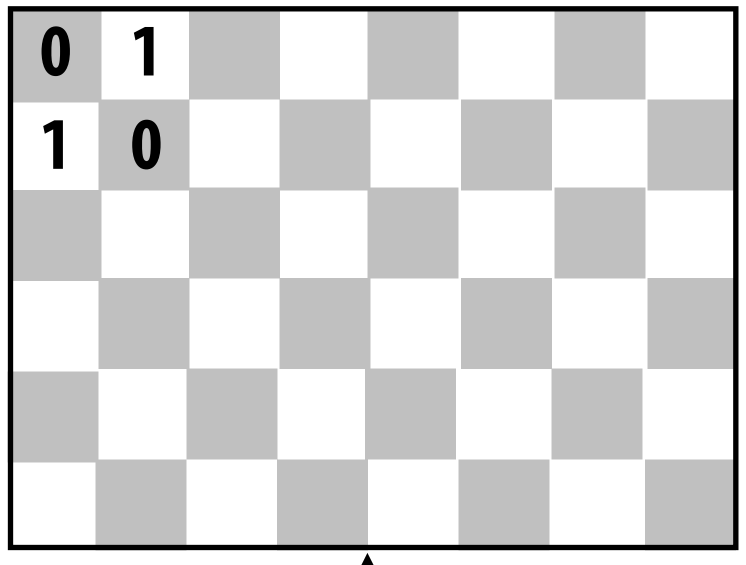
Starting state: draw commands enqueued for pipeline



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

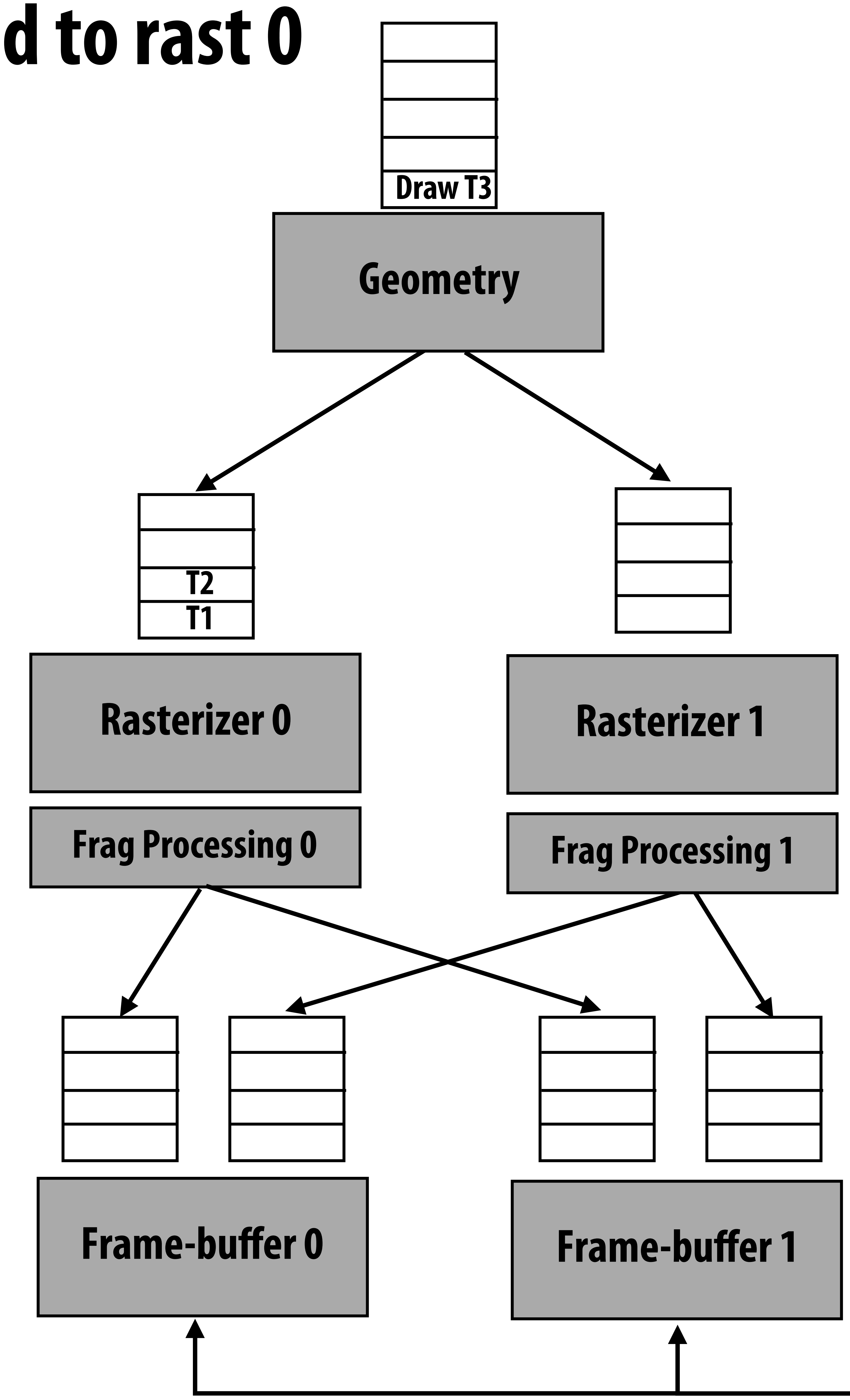


Assume batch size is 2 for assignment to rasterizers.

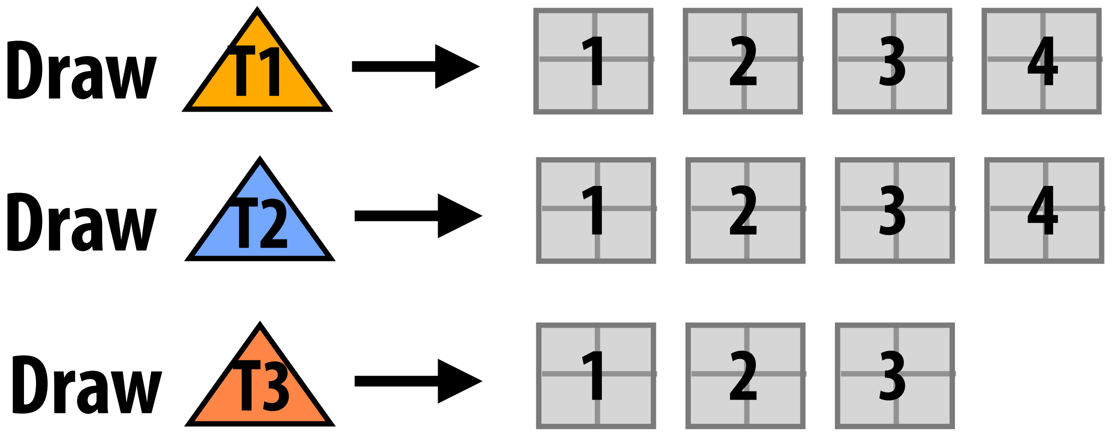


Interleaved render target

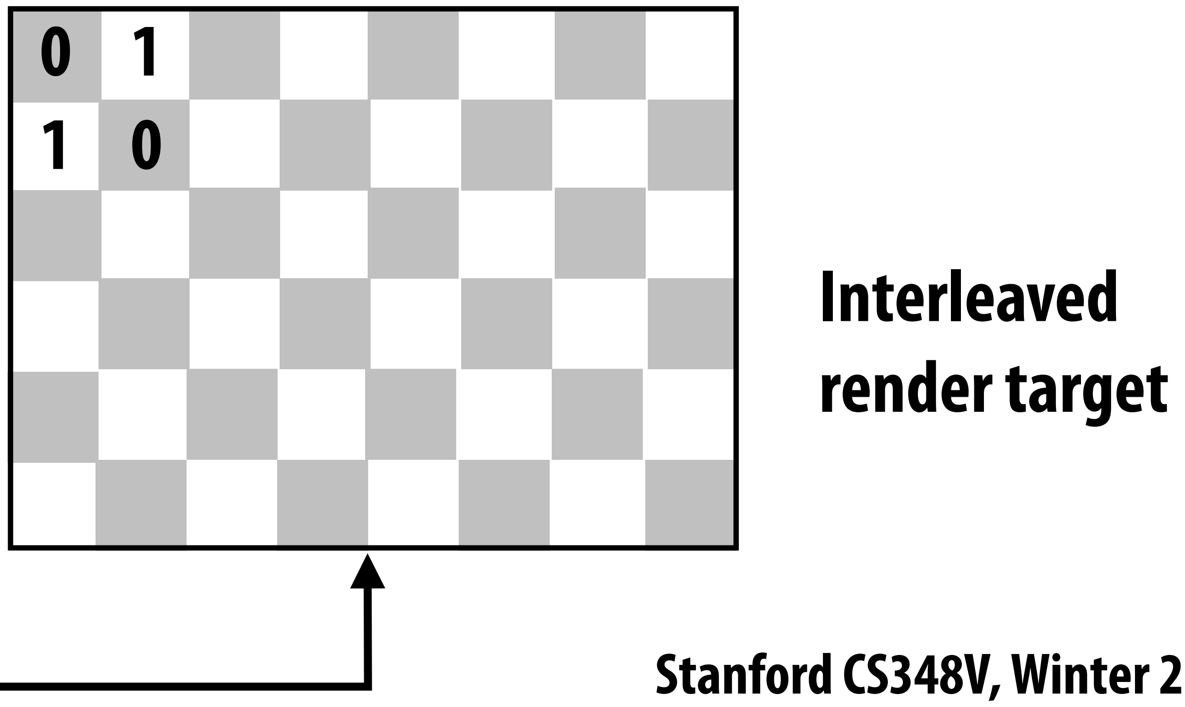
After geometry processing, first two processed triangles assigned to rast 0



Input:

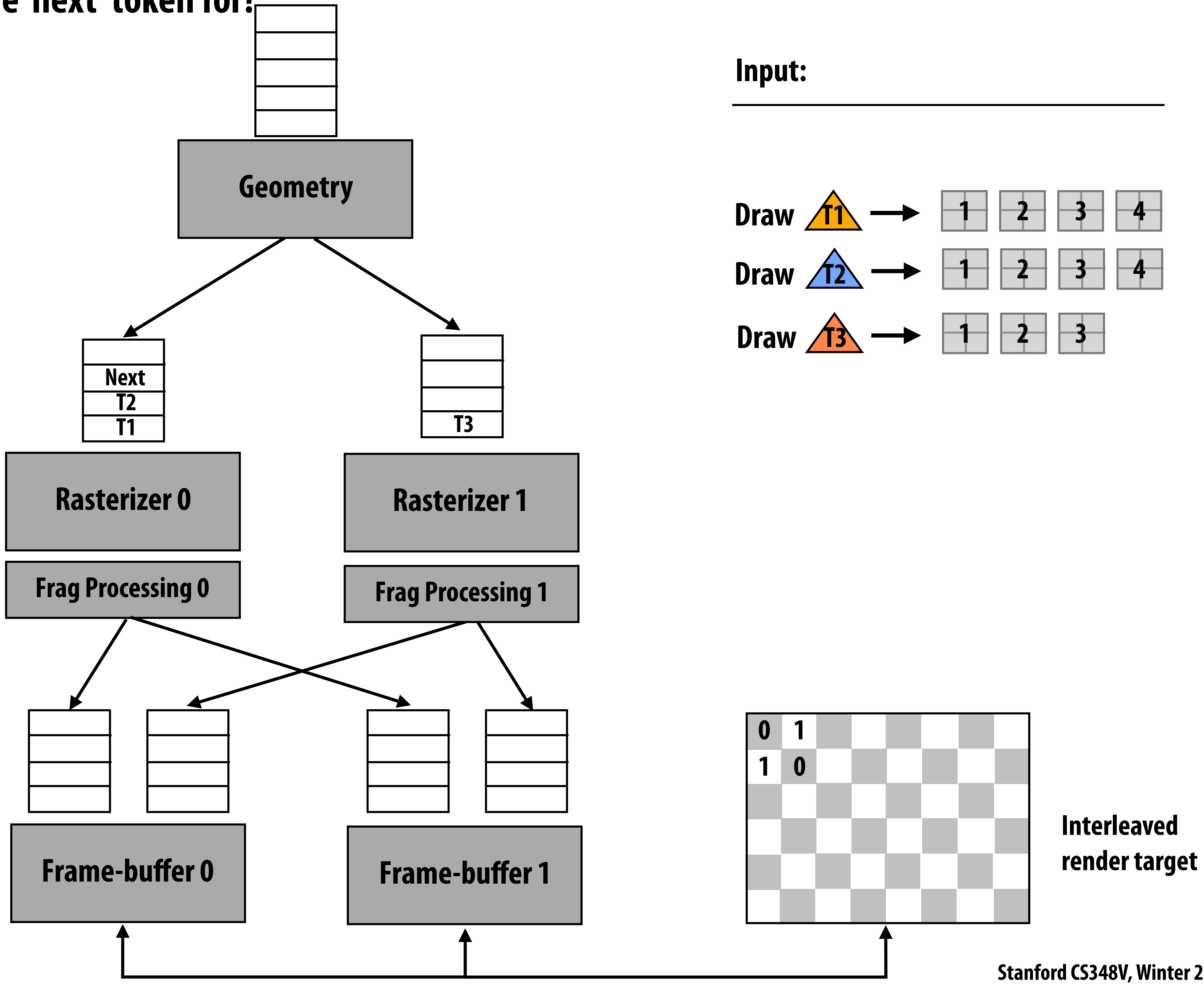


Assume batch size is 2 for assignment to rasterizers.



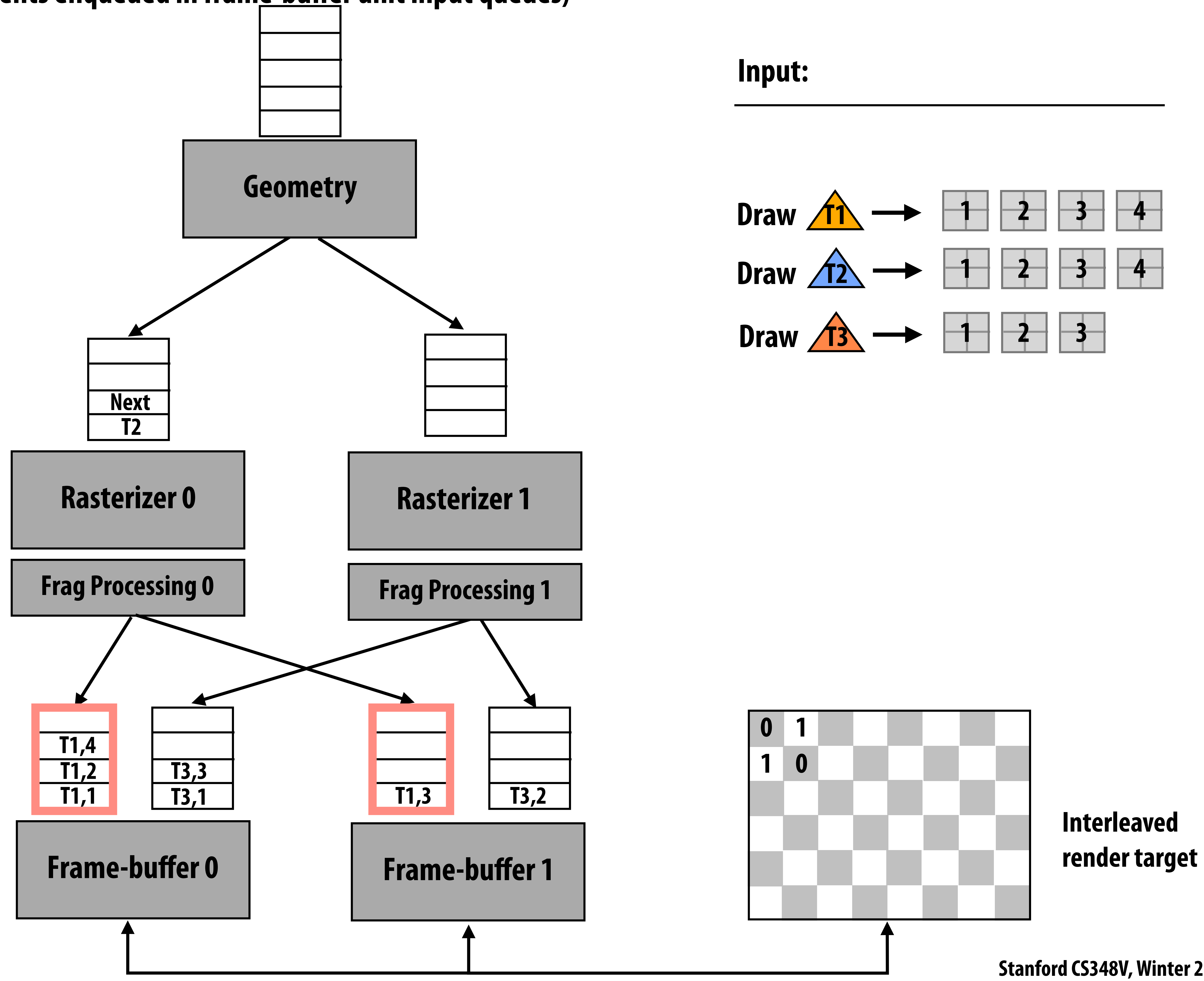
Assign next triangle to rast 1 (round robin policy, batch size = 2)

Q. What is the 'next' token for?



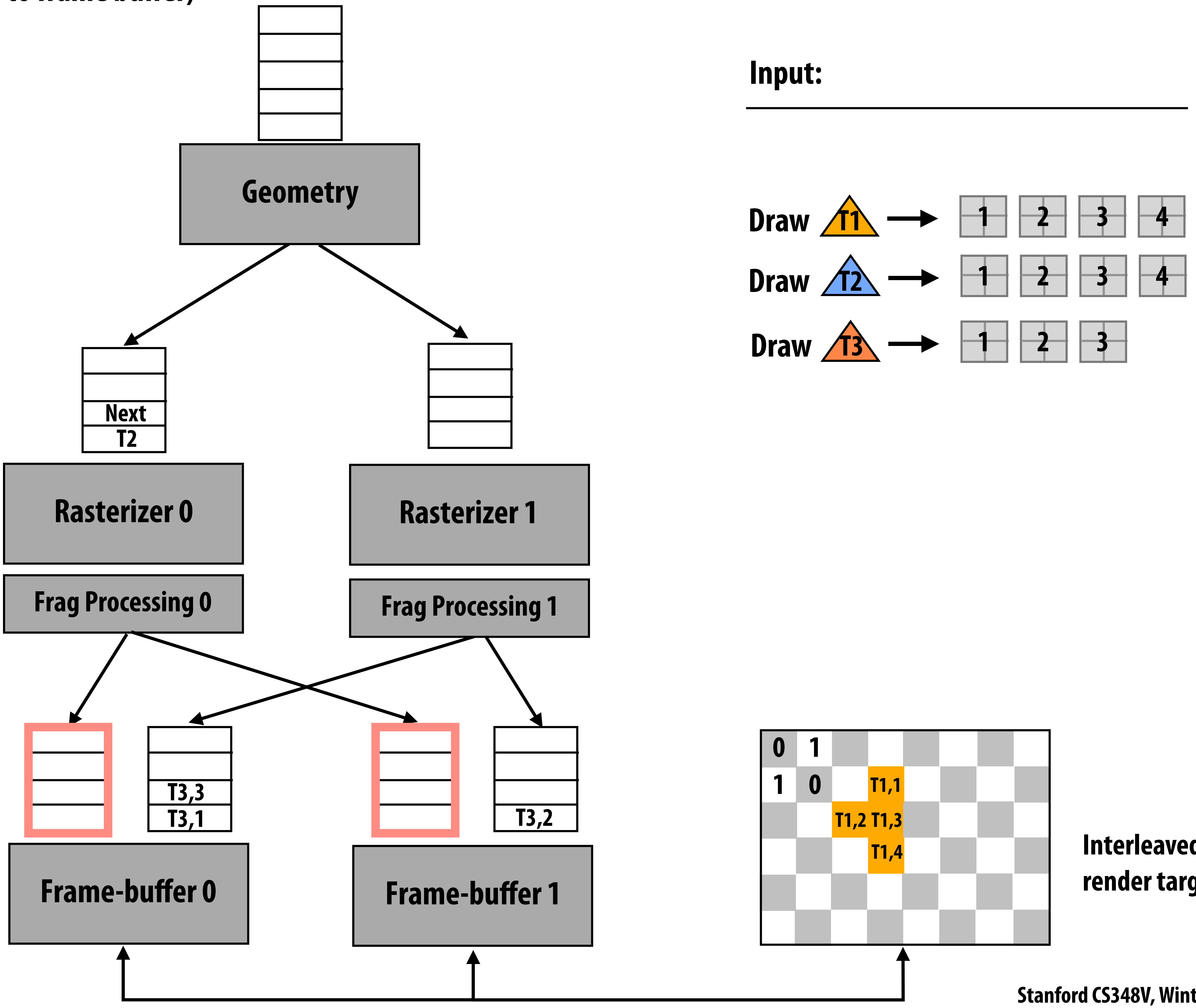
Rast 0 and rast 1 can process T1 and T3 simultaneously

(Shaded fragments enqueued in frame-buffer unit input queues)

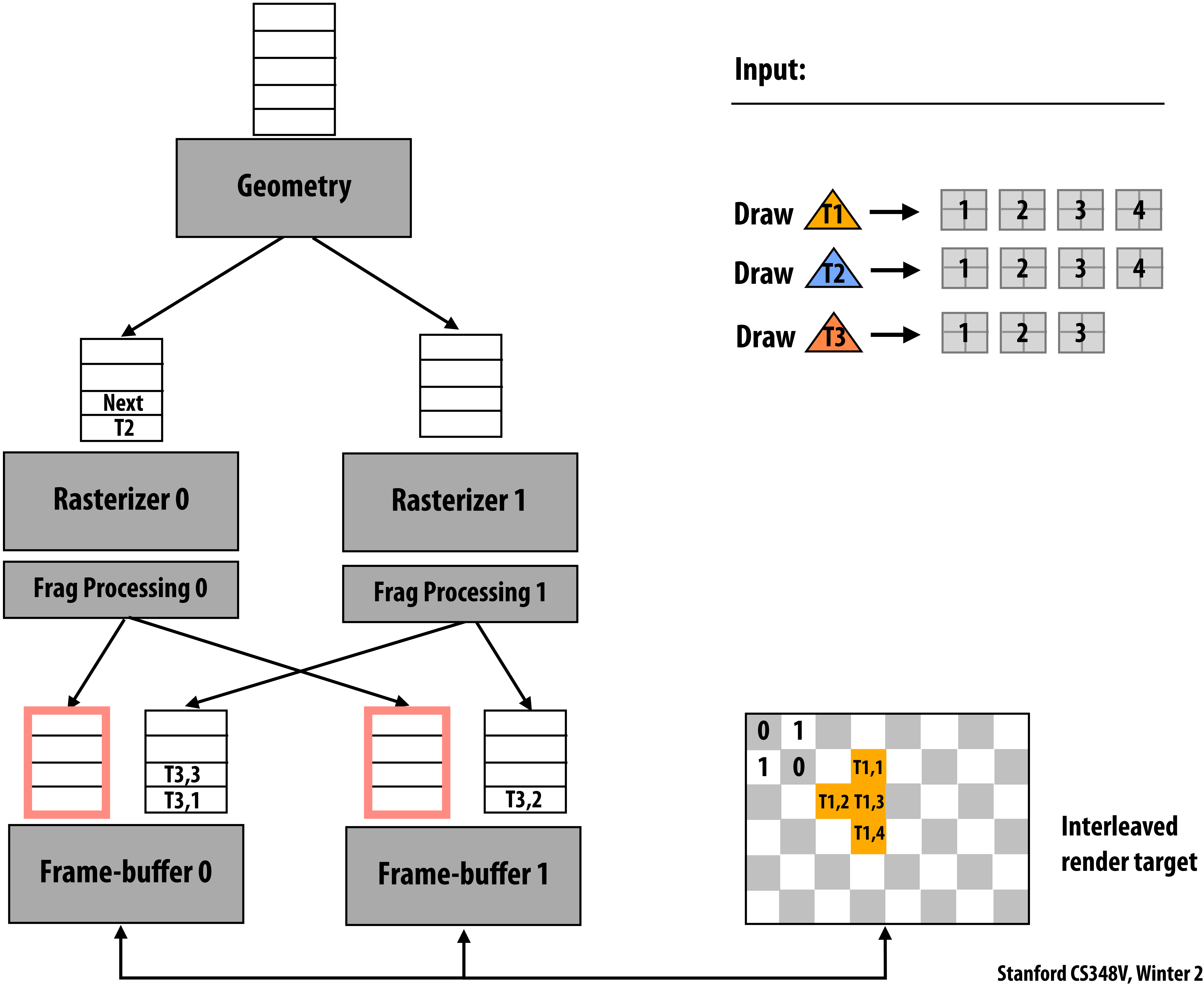


FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)

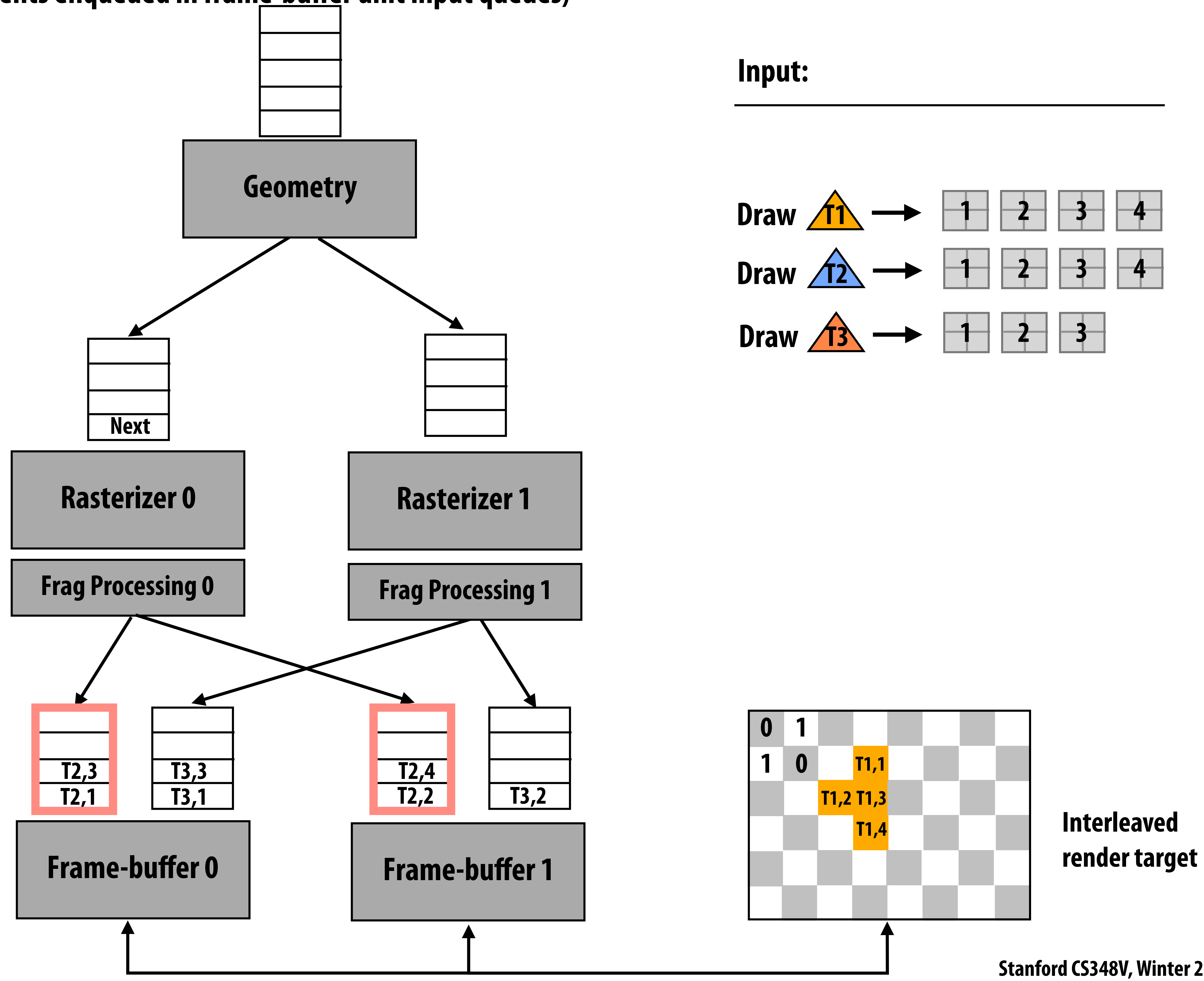


Fragments from T3 cannot be processed yet. Why?

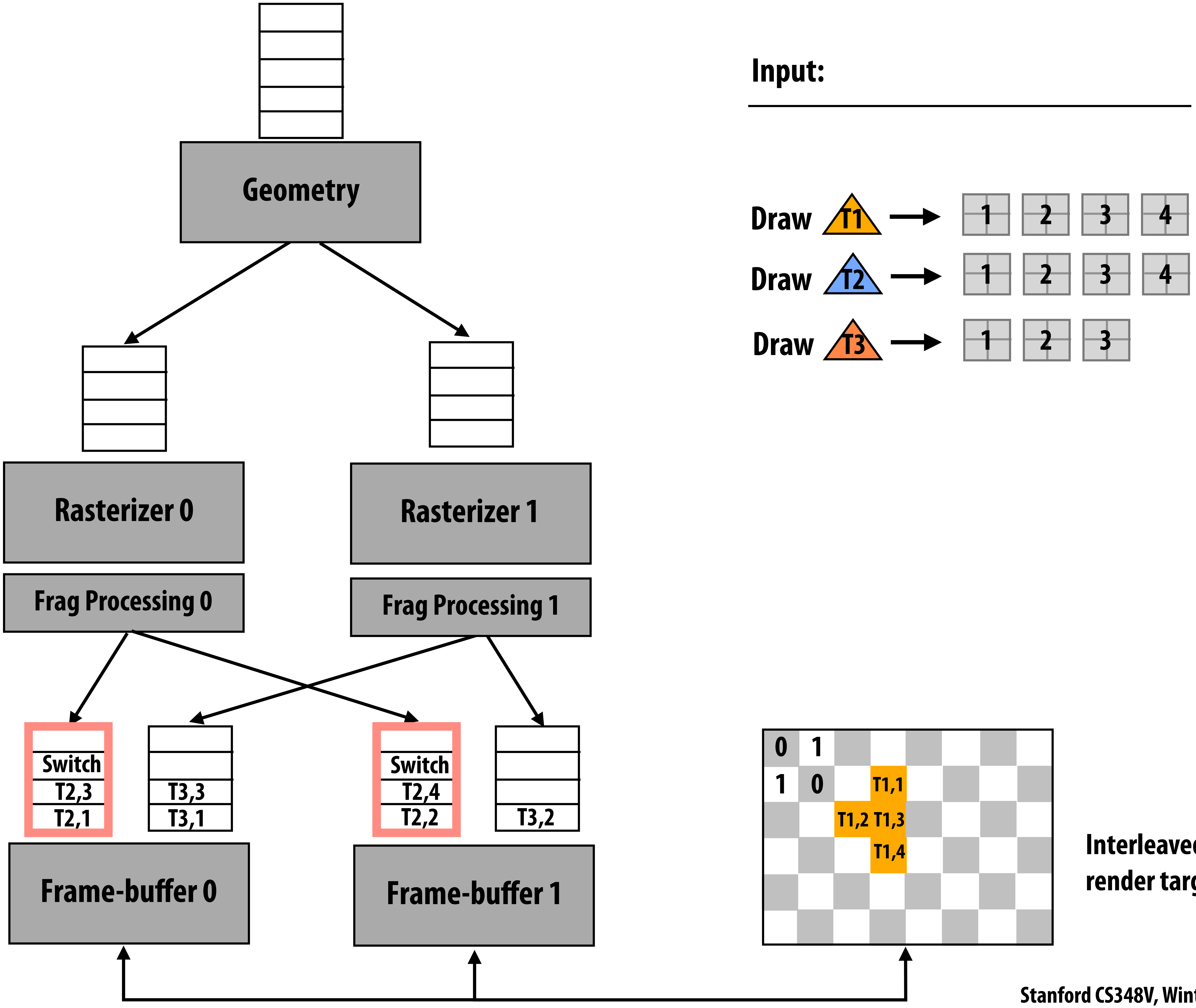


Rast 0 processes T2

(Shaded fragments enqueued in frame-buffer unit input queues)



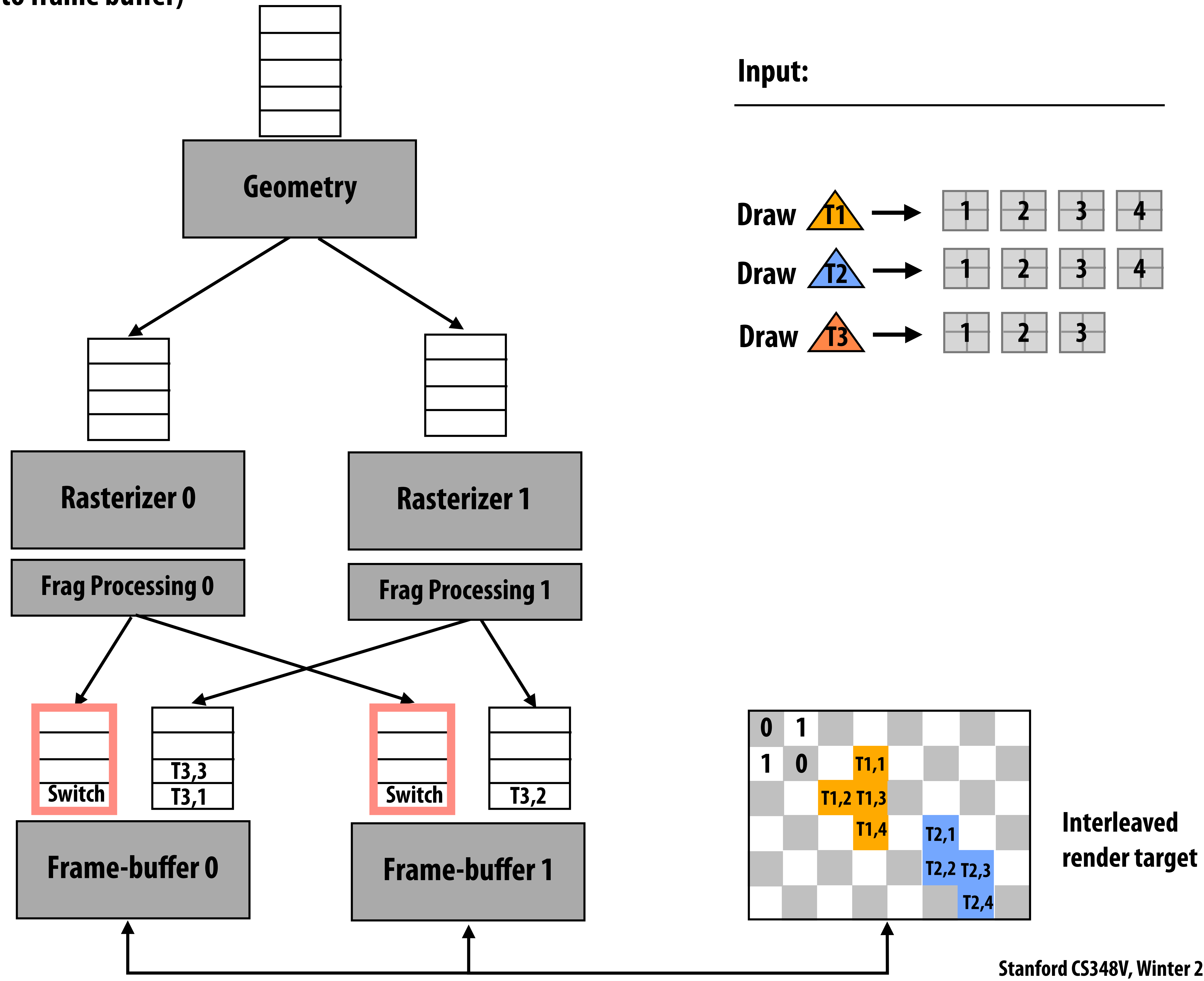
Rast 0 broadcasts 'next' token to all frame-buffer units



Interleaved render target

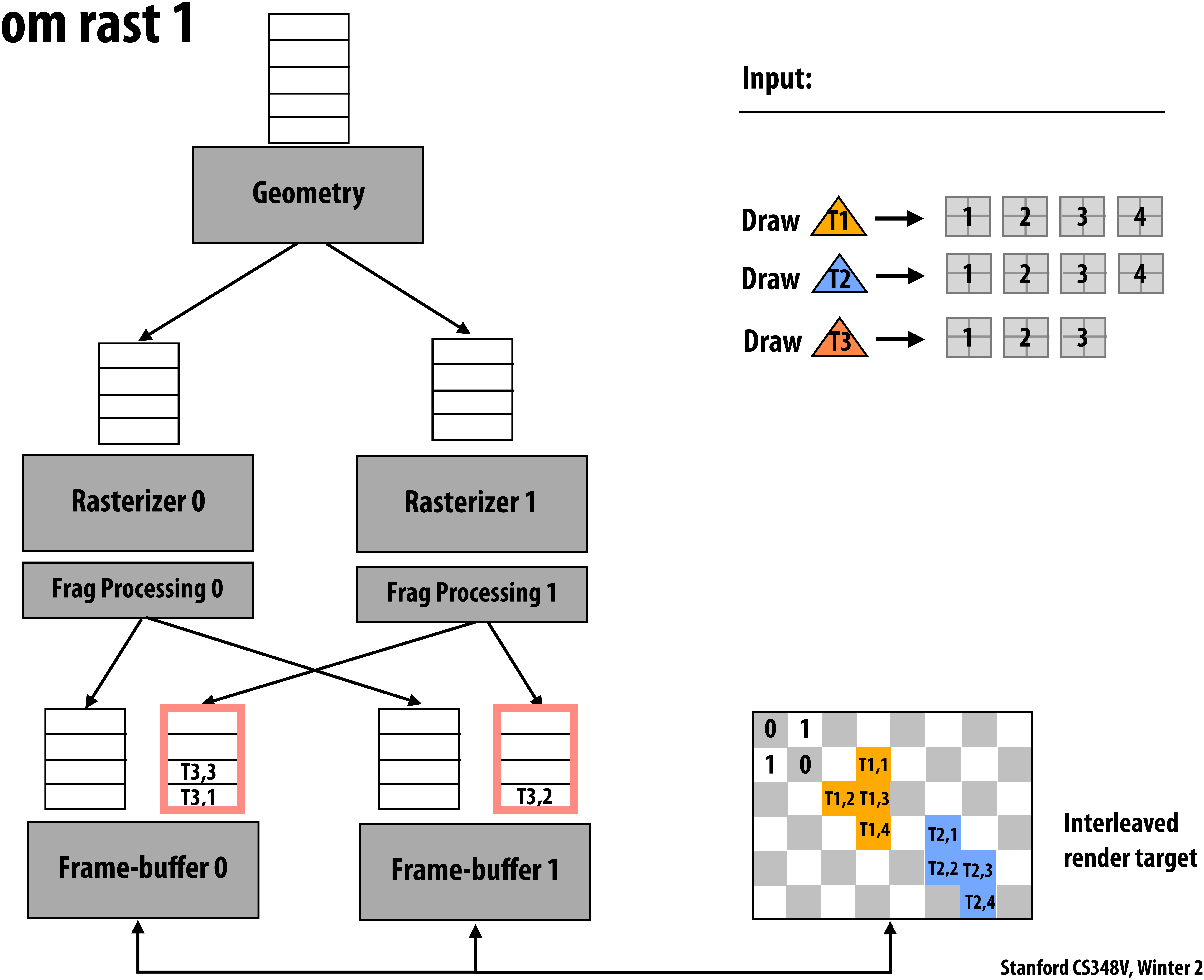
FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)



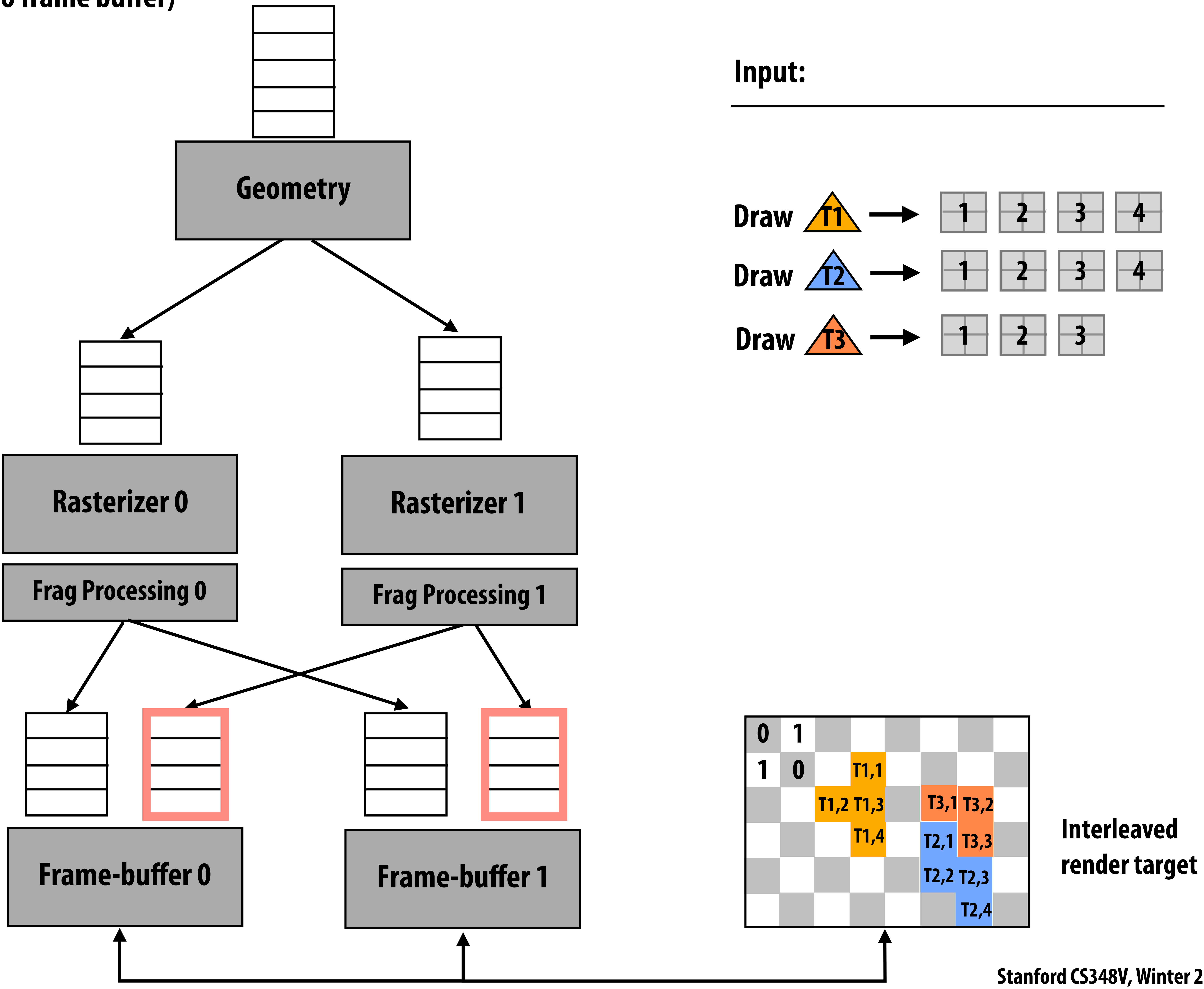
0	1						
1	0		T1,1				
		T1,2	T1,3				
			T1,4		T2,1		
					T2,2	T2,3	
						T2,4	

Switch token reached: frame-buffer units start processing input from rast 1



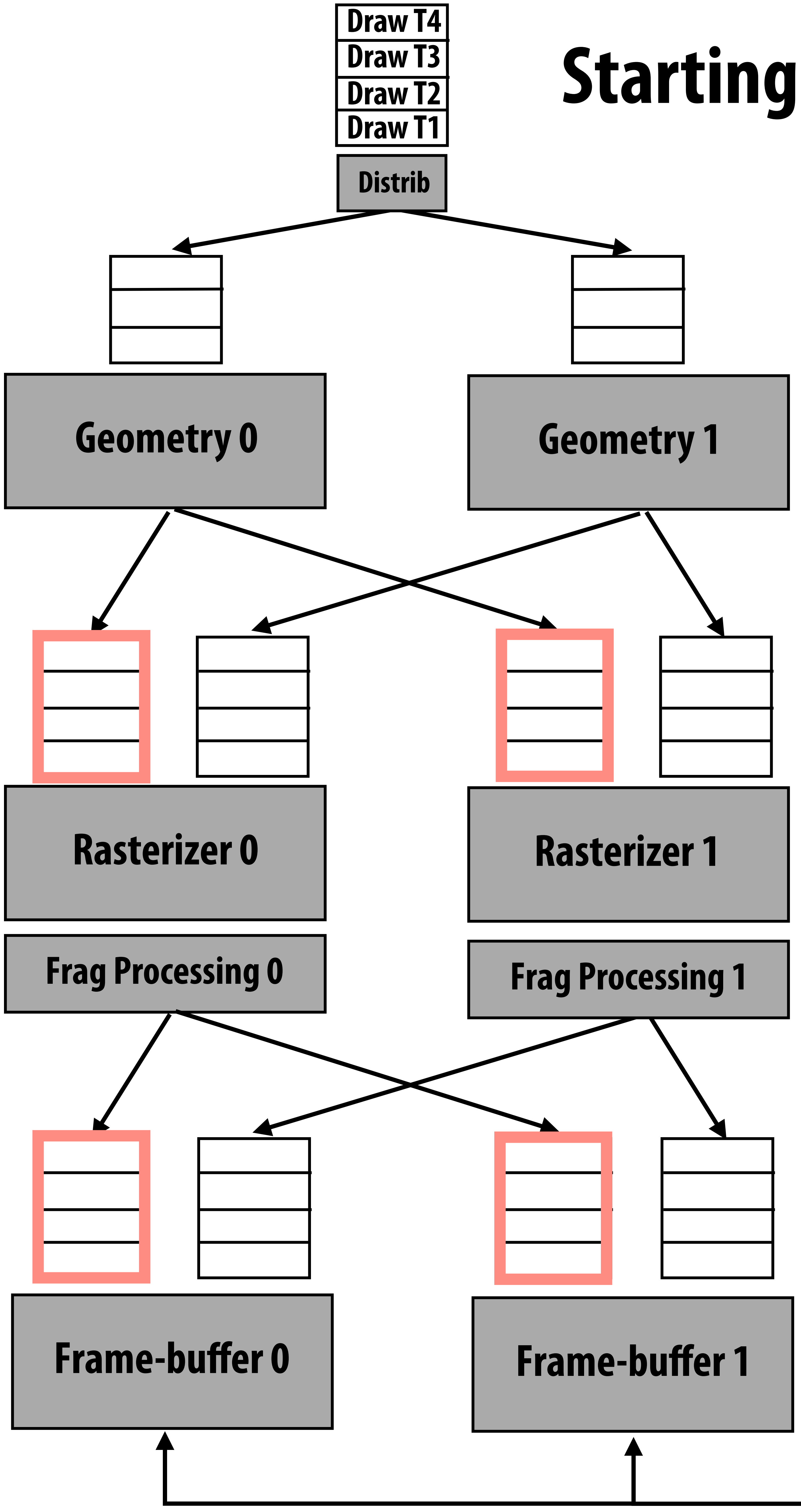
FB 0 and FB 1 can simultaneously process fragments from rast 1

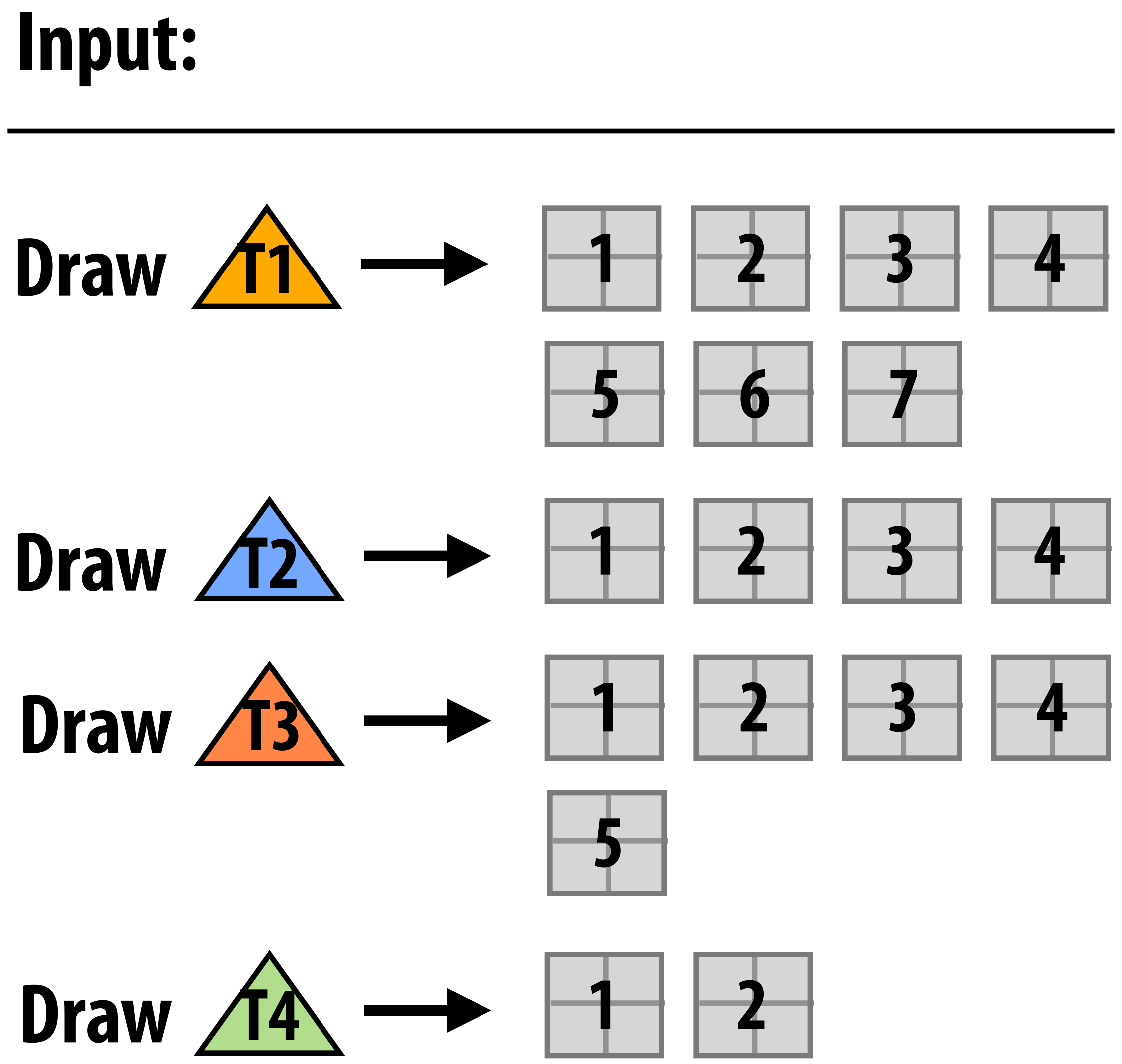
(Notice updates to frame buffer)



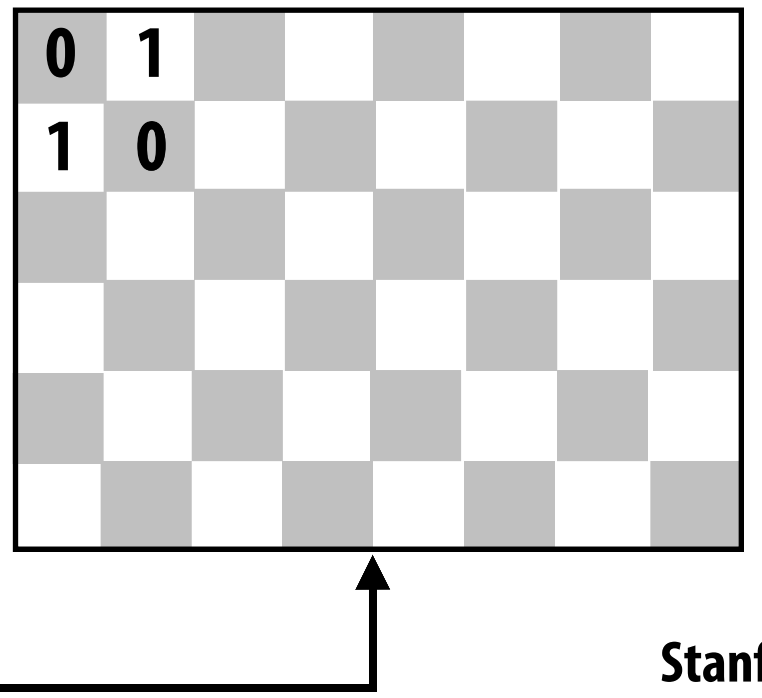
Extending to parallel geometry units

Starting state: commands enqueued



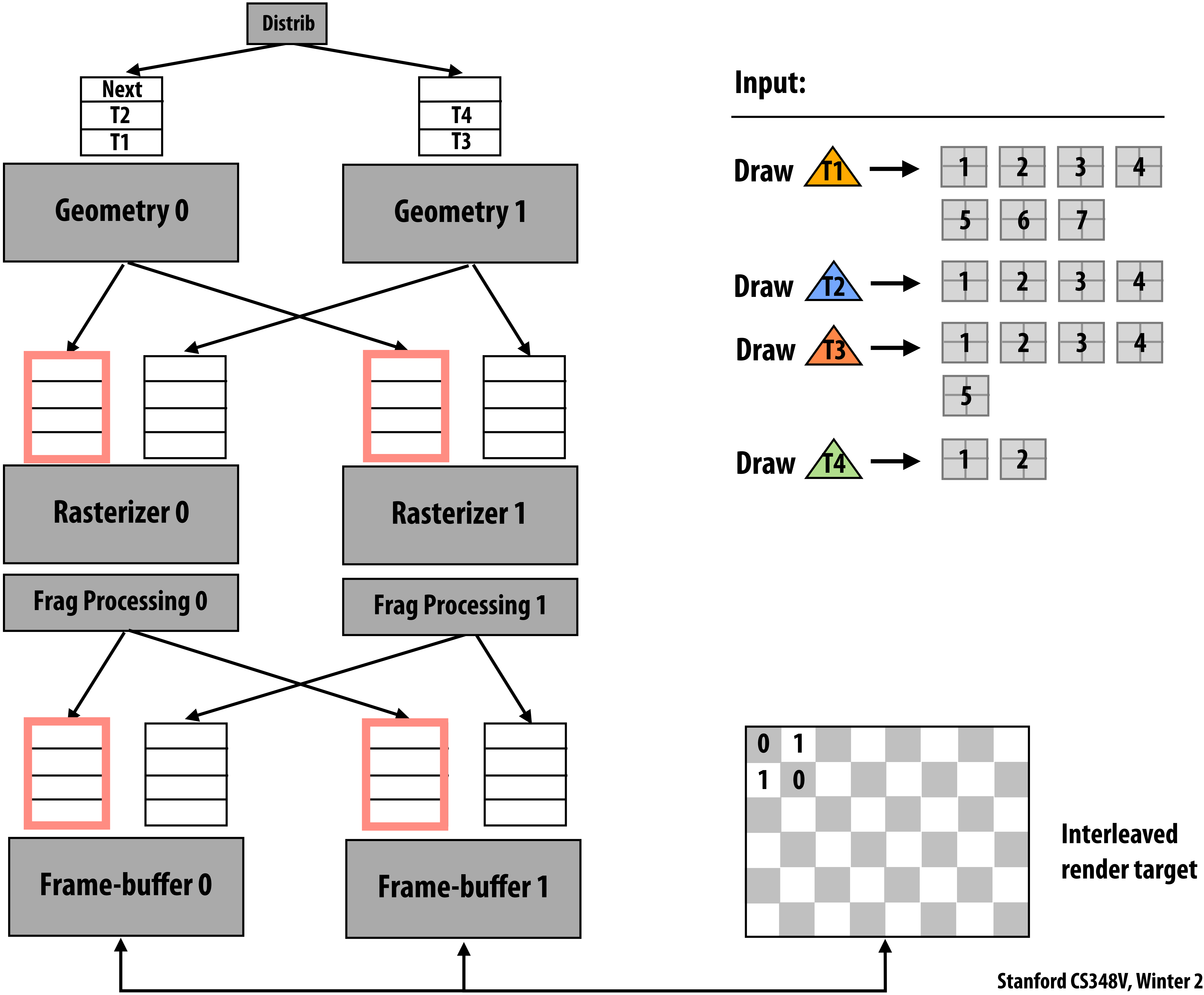


Assume batch size is 2 for assignment to geom units and to rasterizers.



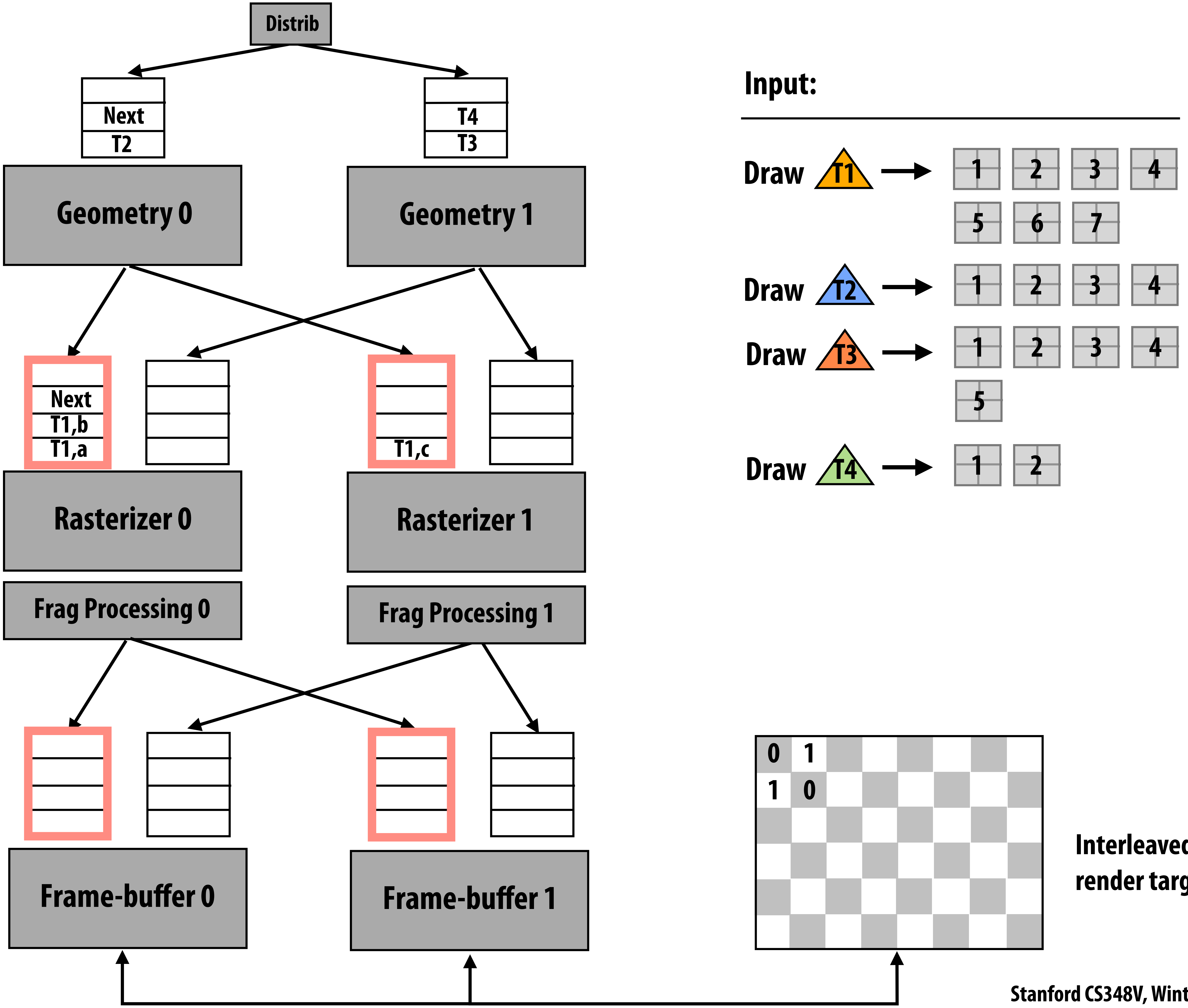
Interleaved render target

Distribute triangles to geom units round-robin (batches of 2)



Geom 0 and geom 1 process triangles in parallel

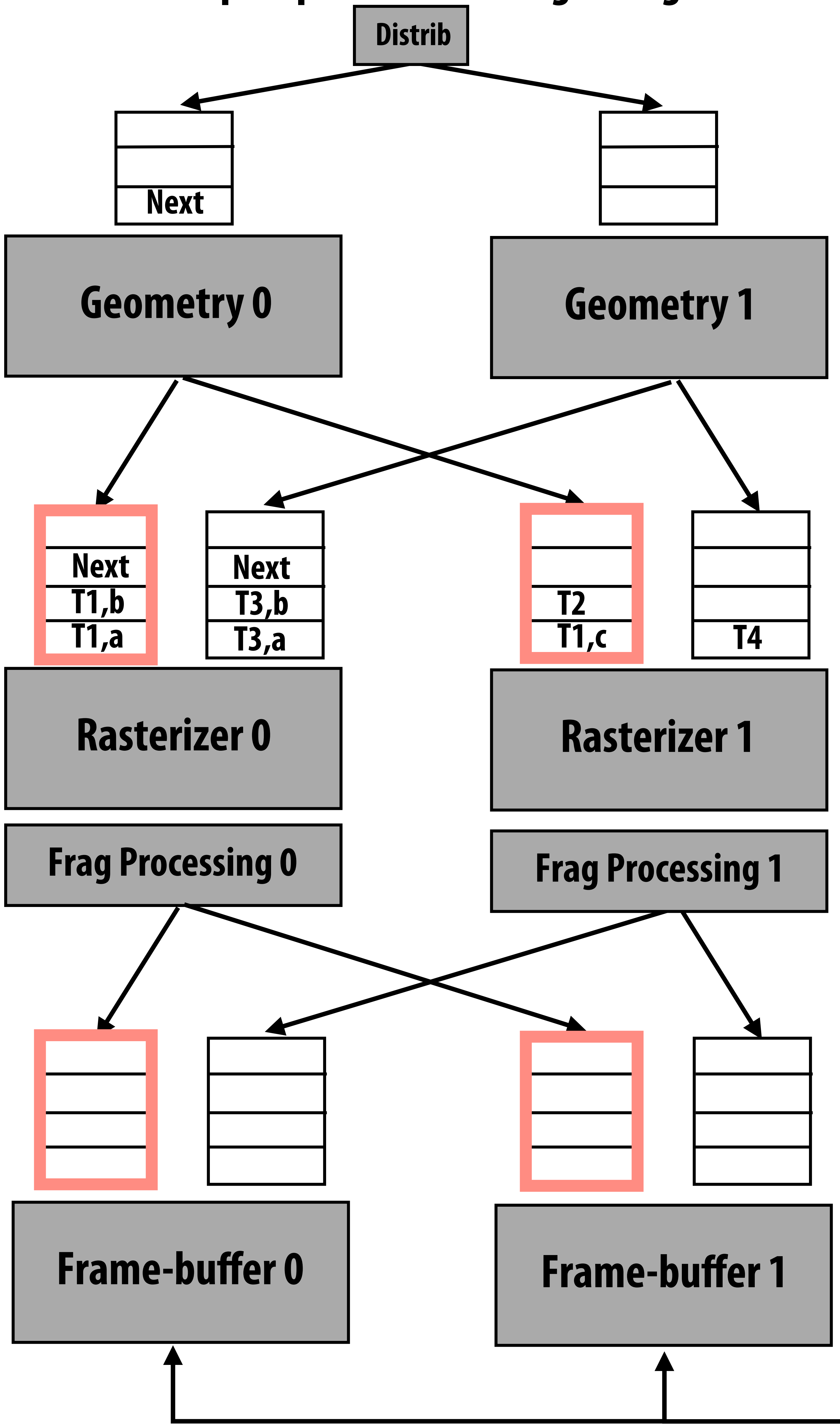
(Results after T1 processed are shown. Note big triangle T1 broken into multiple work items. [Eldridge et al.])

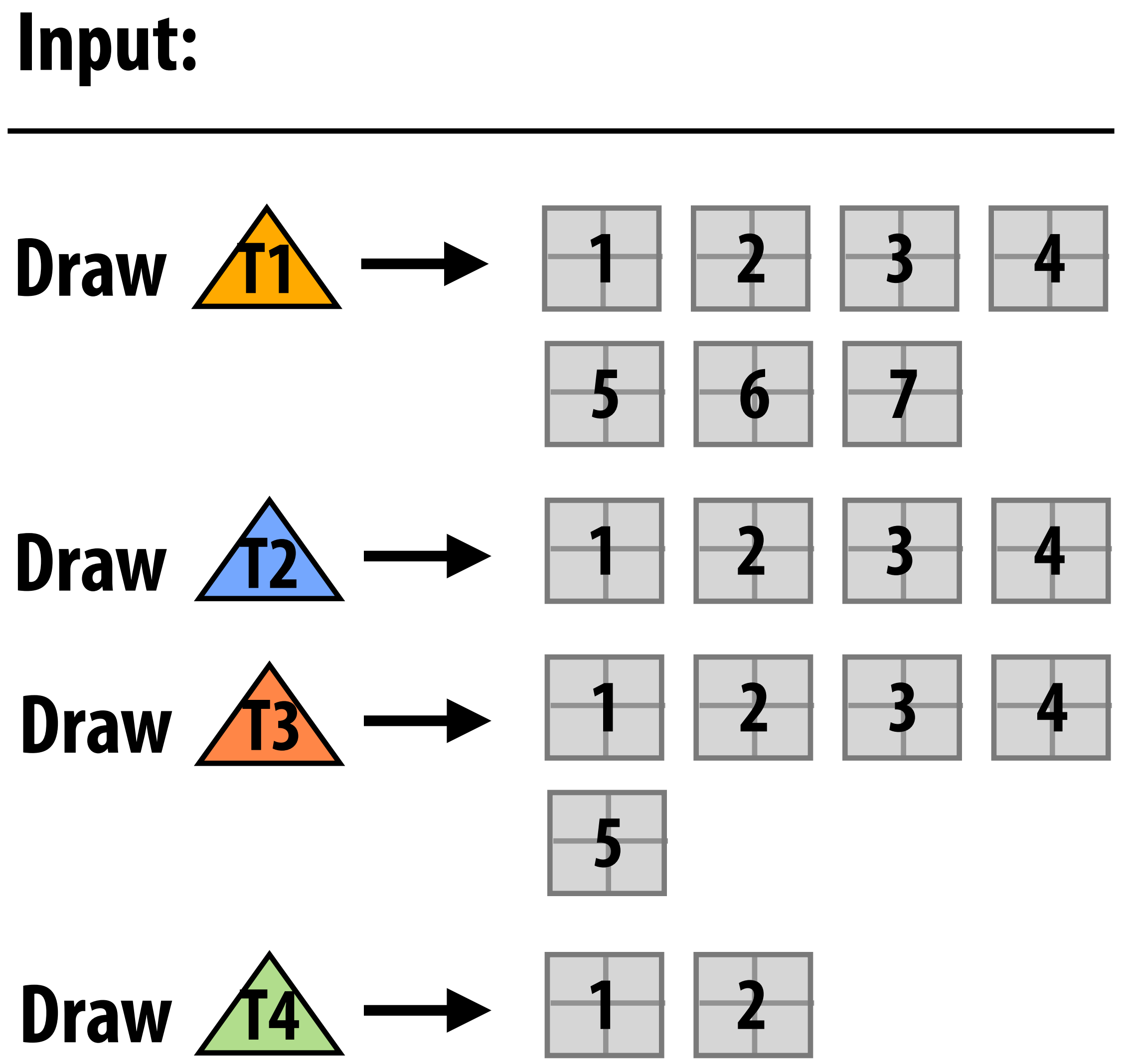


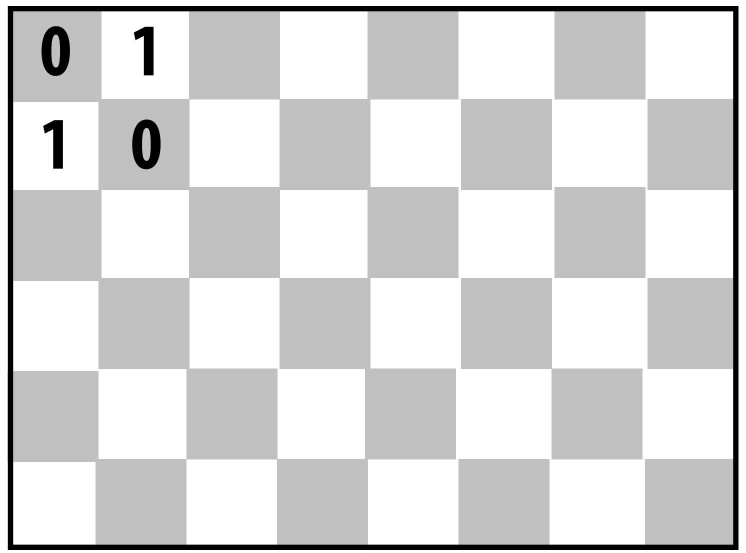
Interleaved render target

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.])

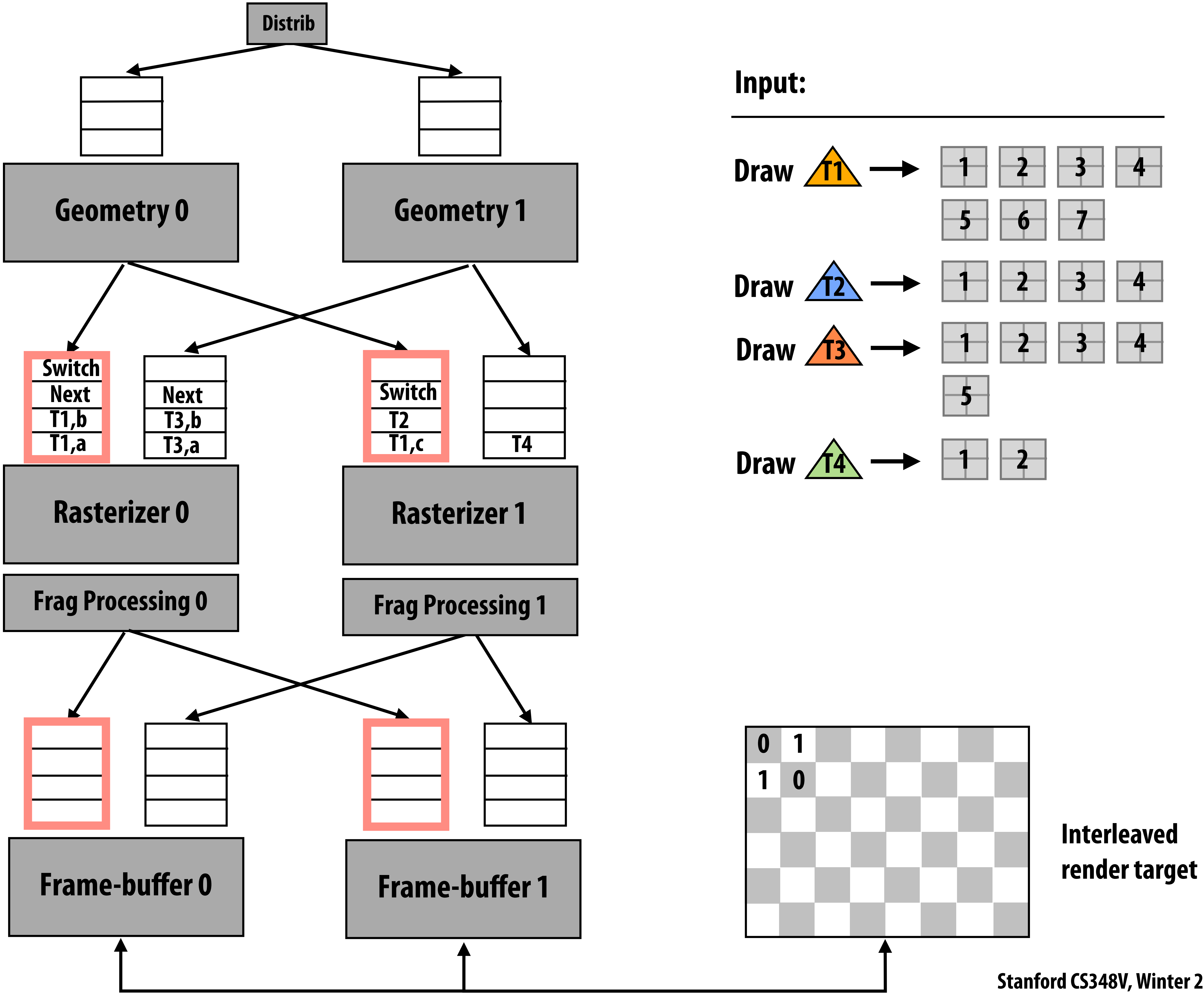






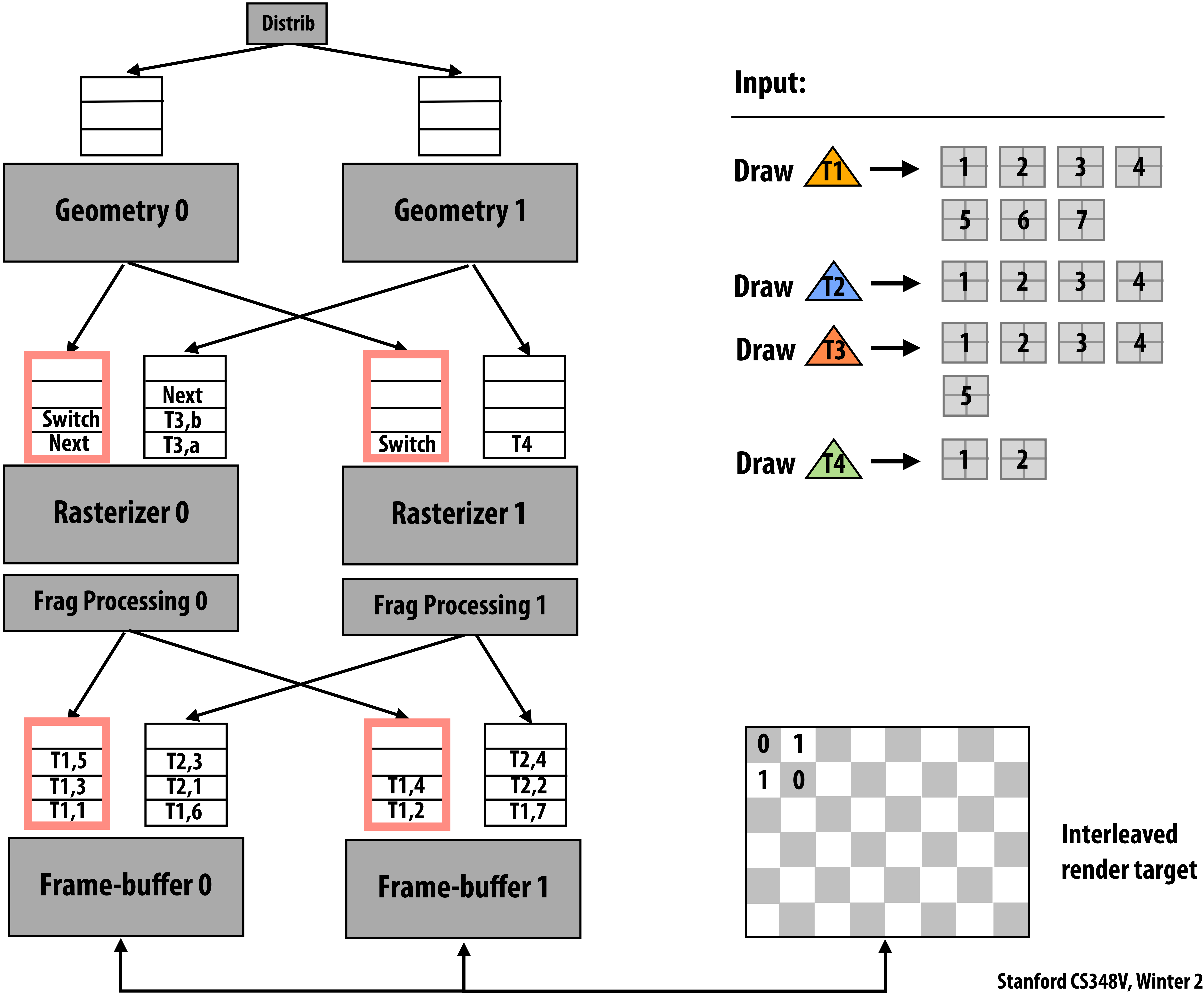
Interleaved render target

Geom 0 broadcasts 'next' token to rasterizers

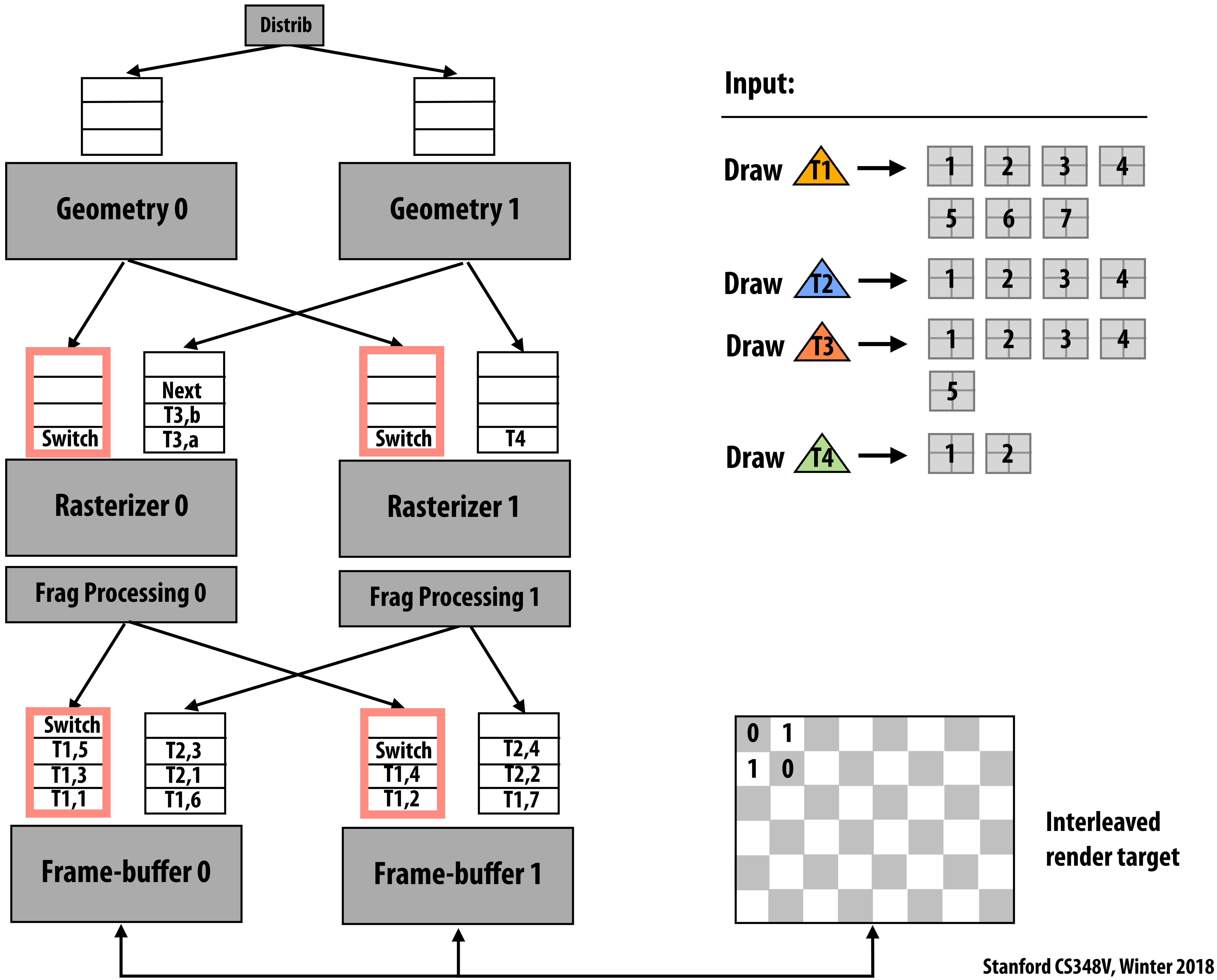


Rast 0 and rast 1 process triangles from geom 0 in parallel

(Shaded fragments enqueued in frame-buffer unit input queues)

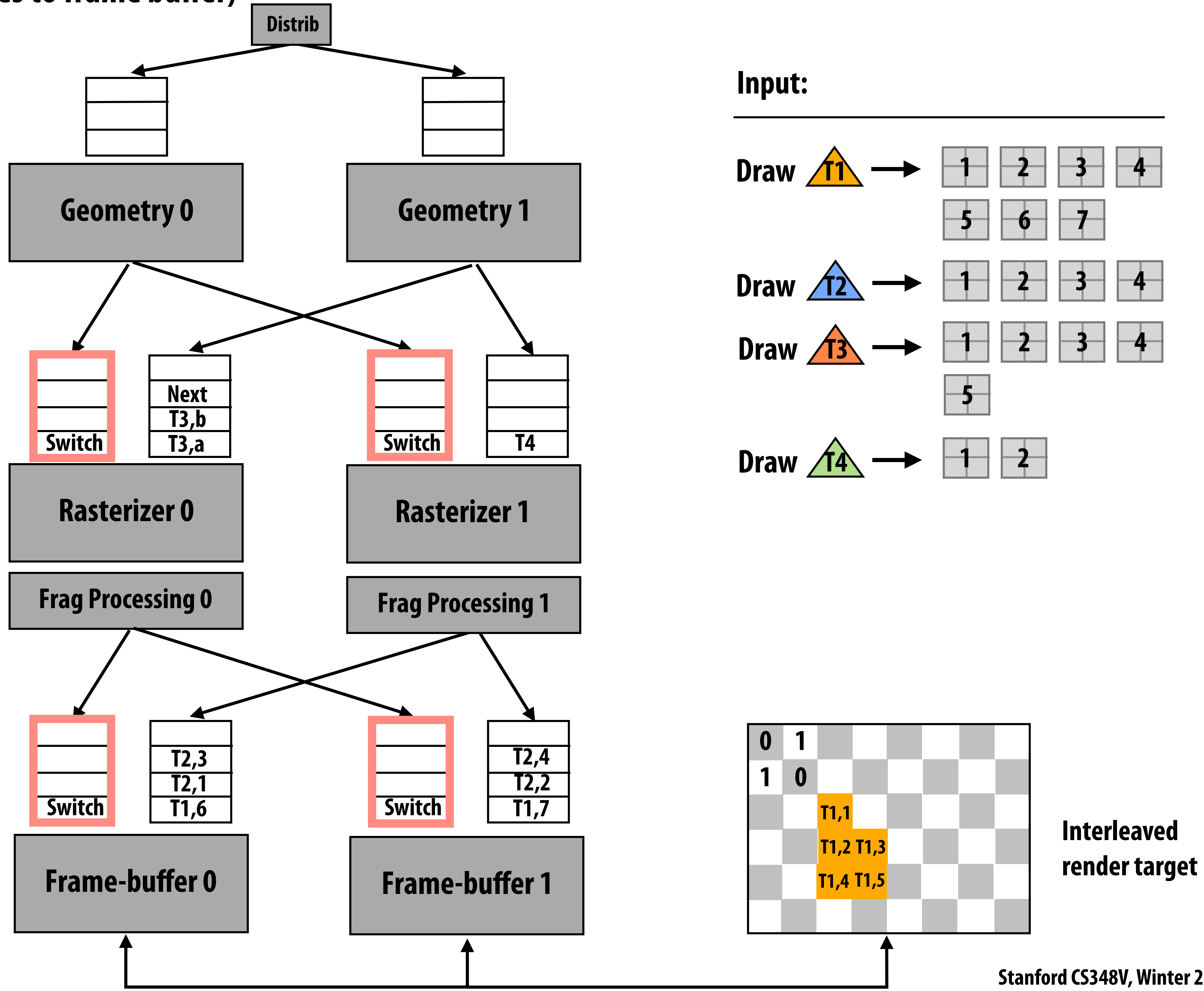


Rast 0 broadcasts 'next' token to FB units (end of geom 0, rast 0)

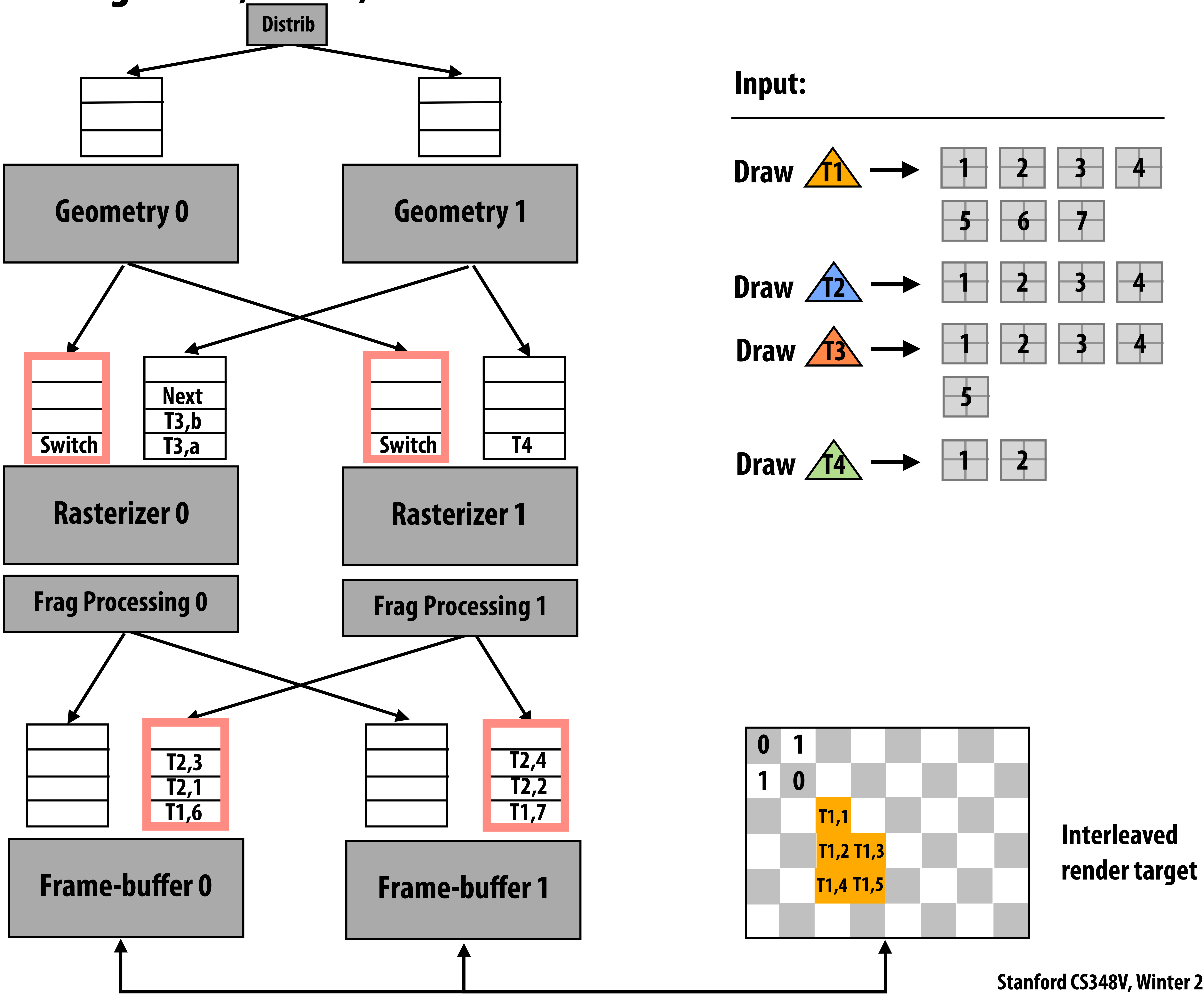


Frame-buffer units process frags from (geom 0, rast 0) in parallel

(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)



Distrib

Geometry 0

Geometry 1

Switch

Next
T3,b
T3,a

Switch

T4

Rasterizer 0

Rasterizer 1

Frag Processing 0

Frag Processing 1

T2,3
T2,1
T1,6

T2,4
T2,2
T1,7

Frame-buffer 0

Frame-buffer 1

Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

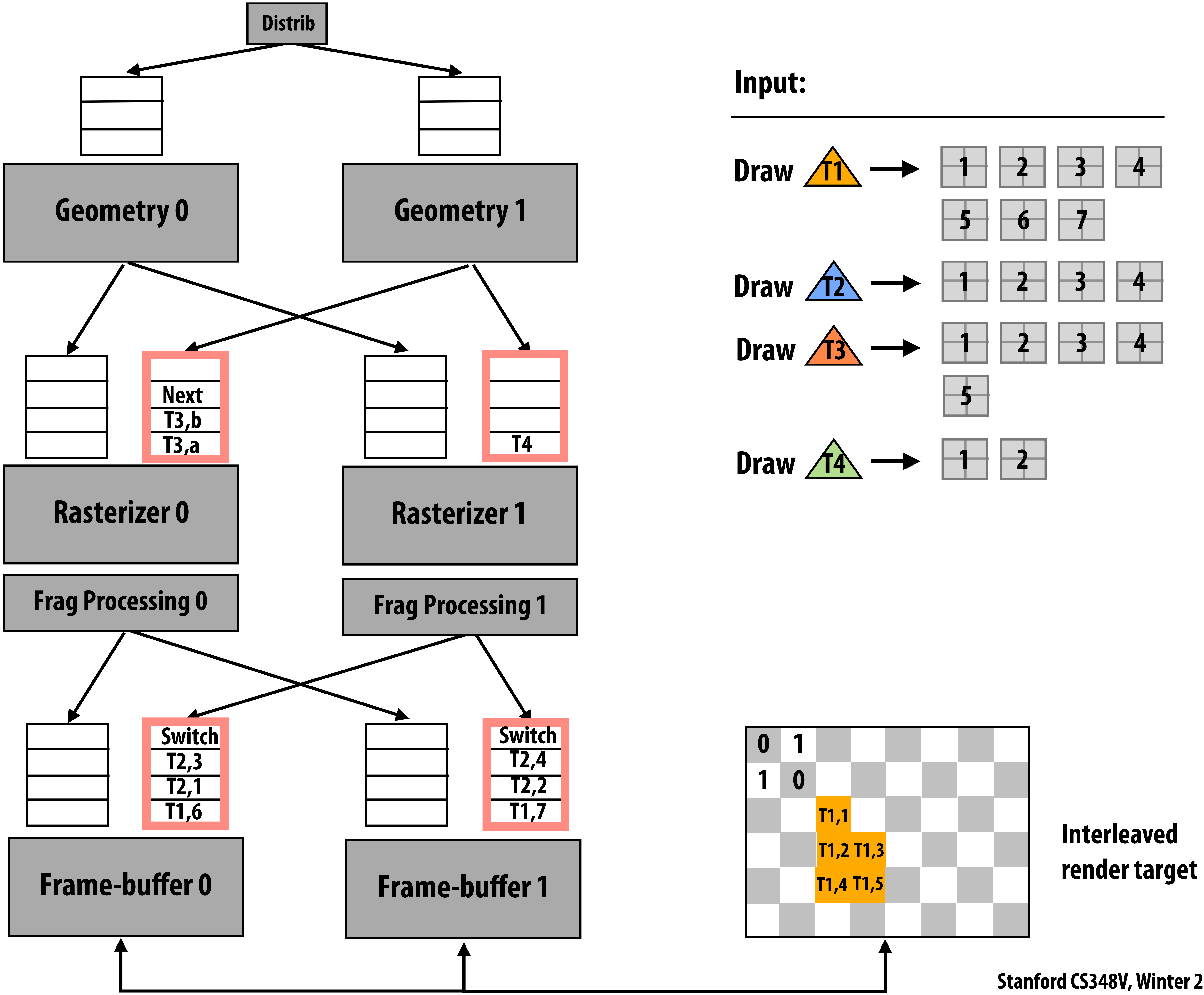
Draw T4 → 1 2

0 1
1 0

T1,1
T1,2 T1,3
T1,4 T1,5

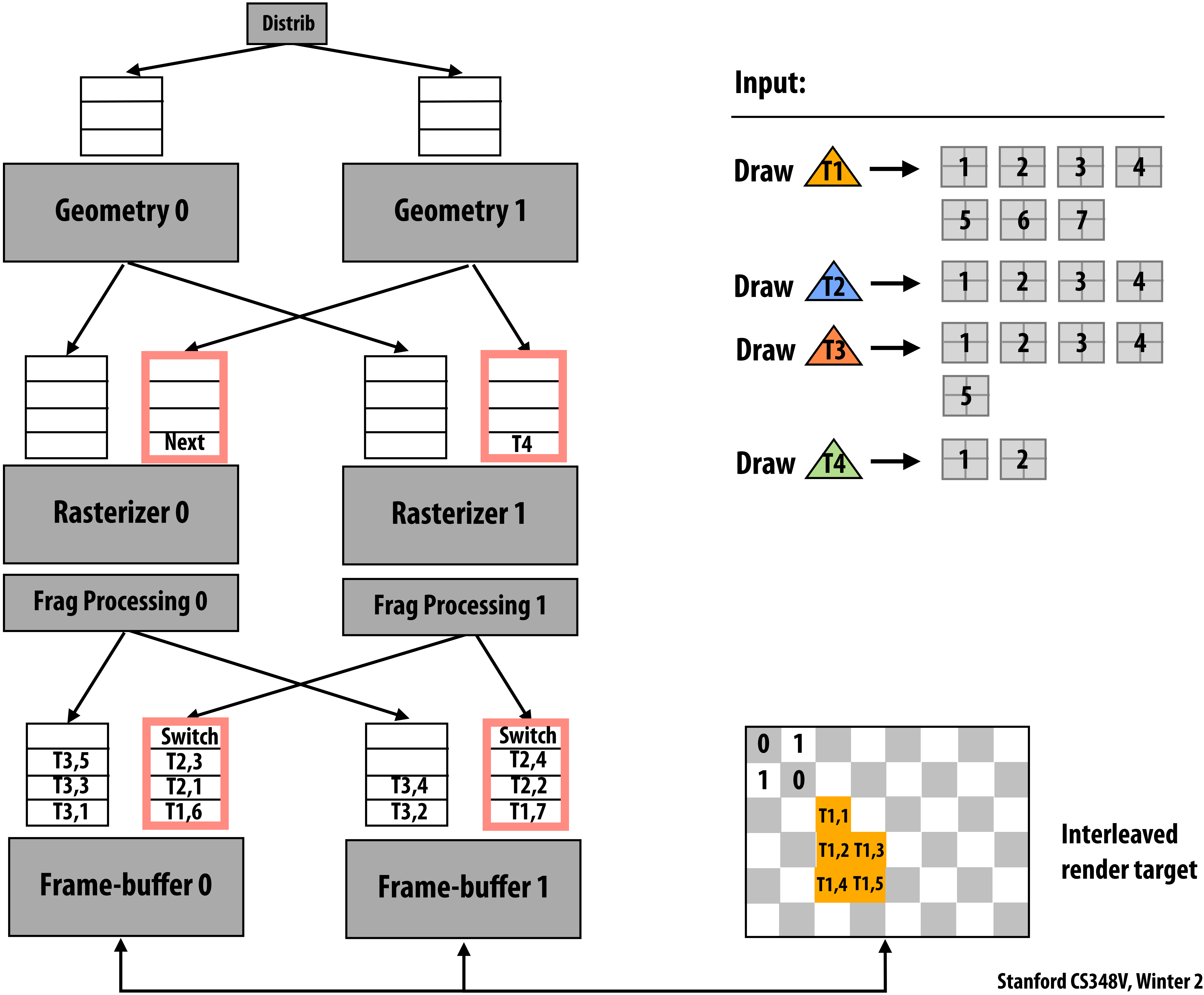
Interleaved render target

"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)

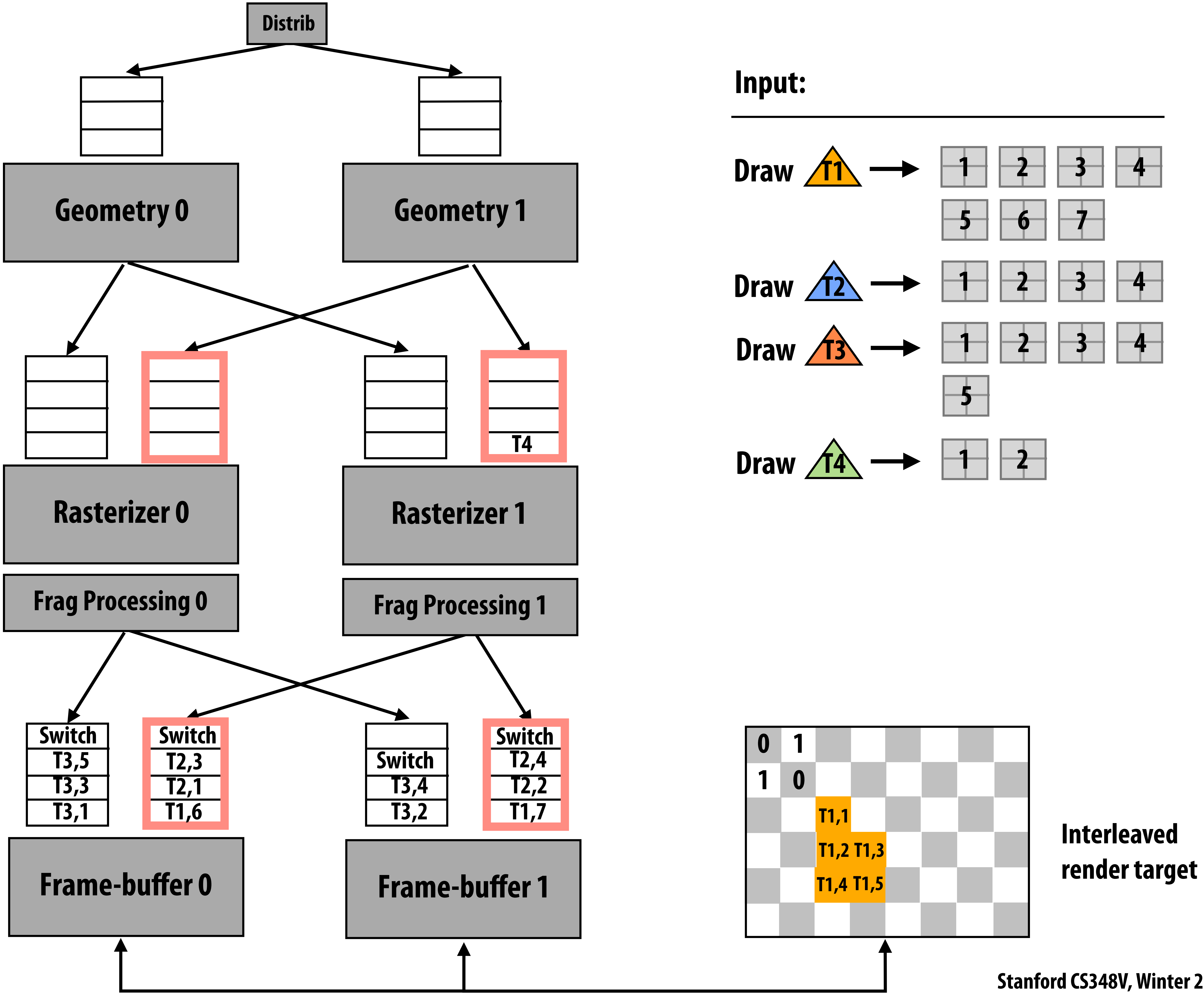


Rast 0 processes triangles from geom 1

(Note Rast 1 has work to do, but cannot make progress because its output queues are full)

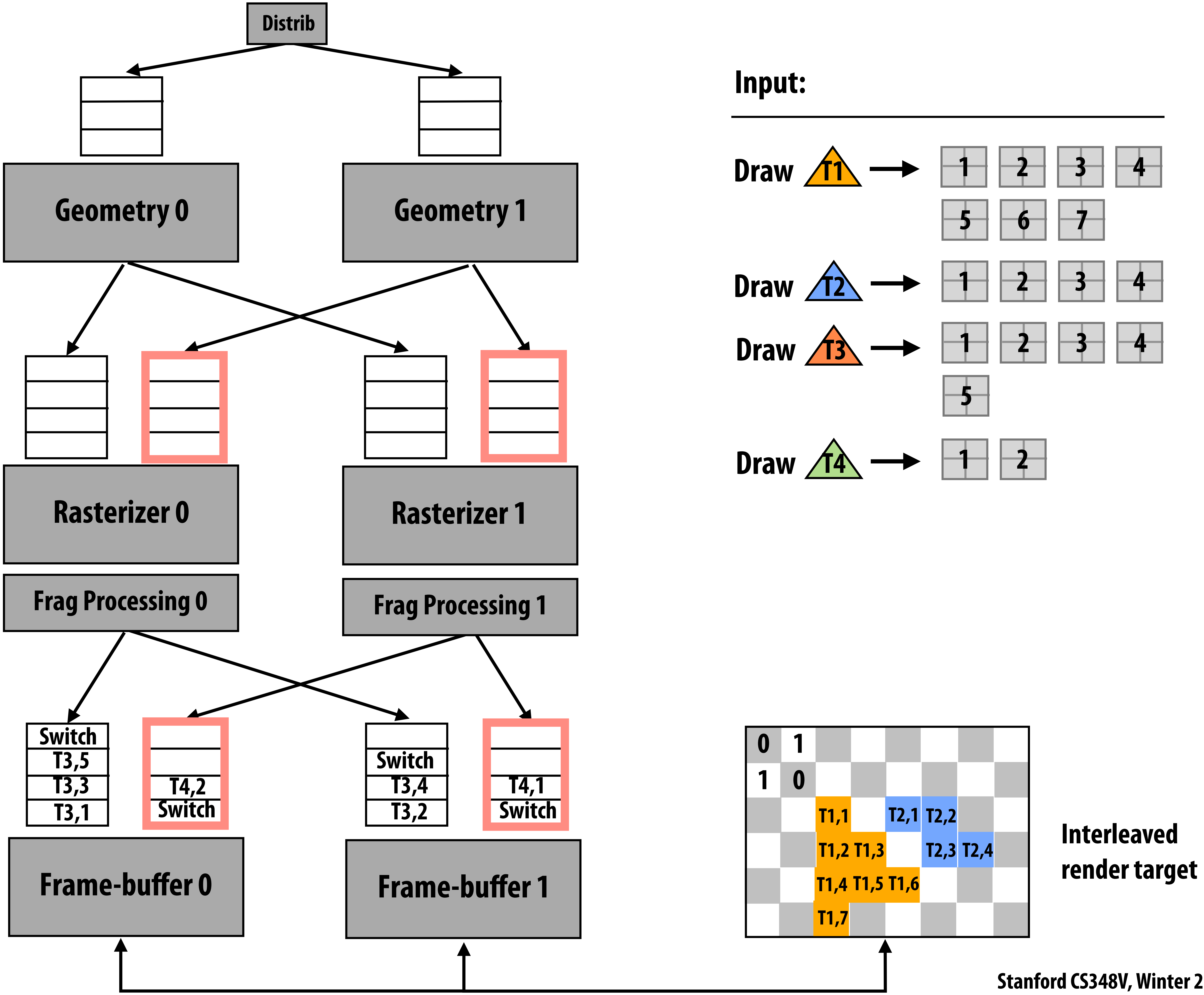


Rast 0 broadcasts “end of geom 1, rast 0” token to frame-buffer units

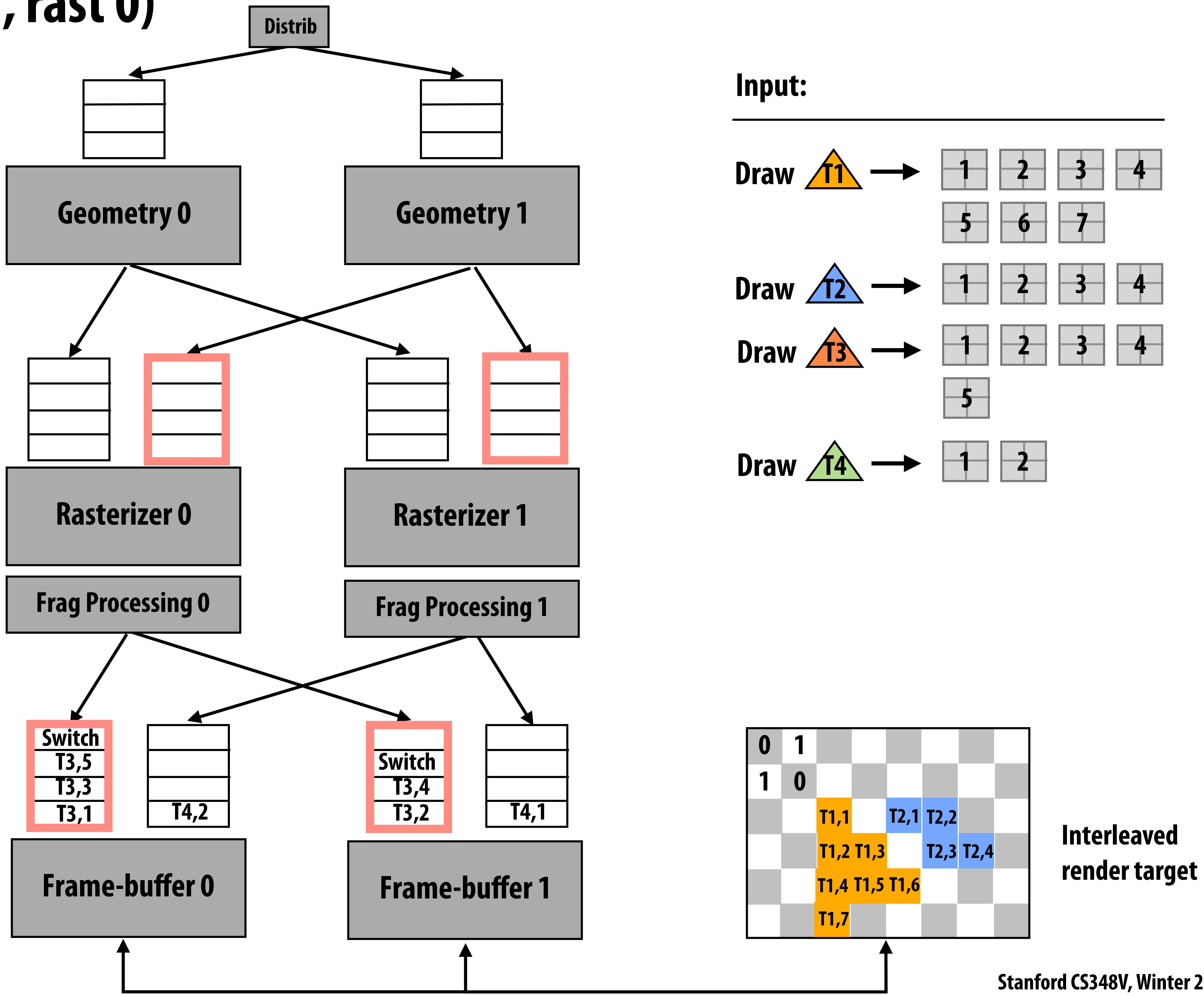


Frame-buffer units process frags from (geom 0, rast 1) in parallel

(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:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

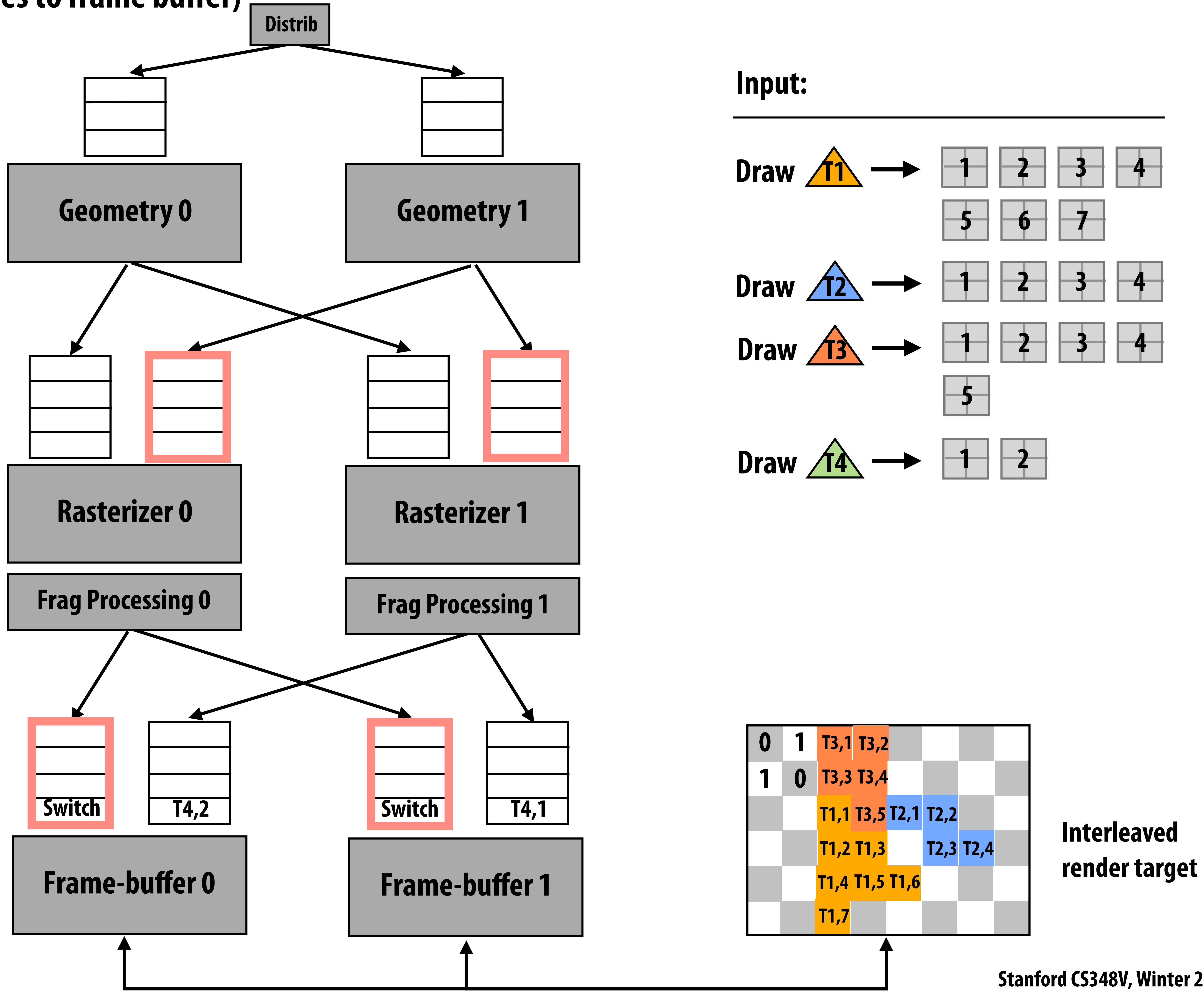
Draw T4 → 1 2

0	1						
1	0						
		T1,1		T2,1	T2,2		
		T1,2	T1,3		T2,3	T2,4	
		T1,4	T1,5	T1,6			
		T1,7					

Interleaved render target

Frame-buffer units process frags from (geom 1, rast 0) in parallel

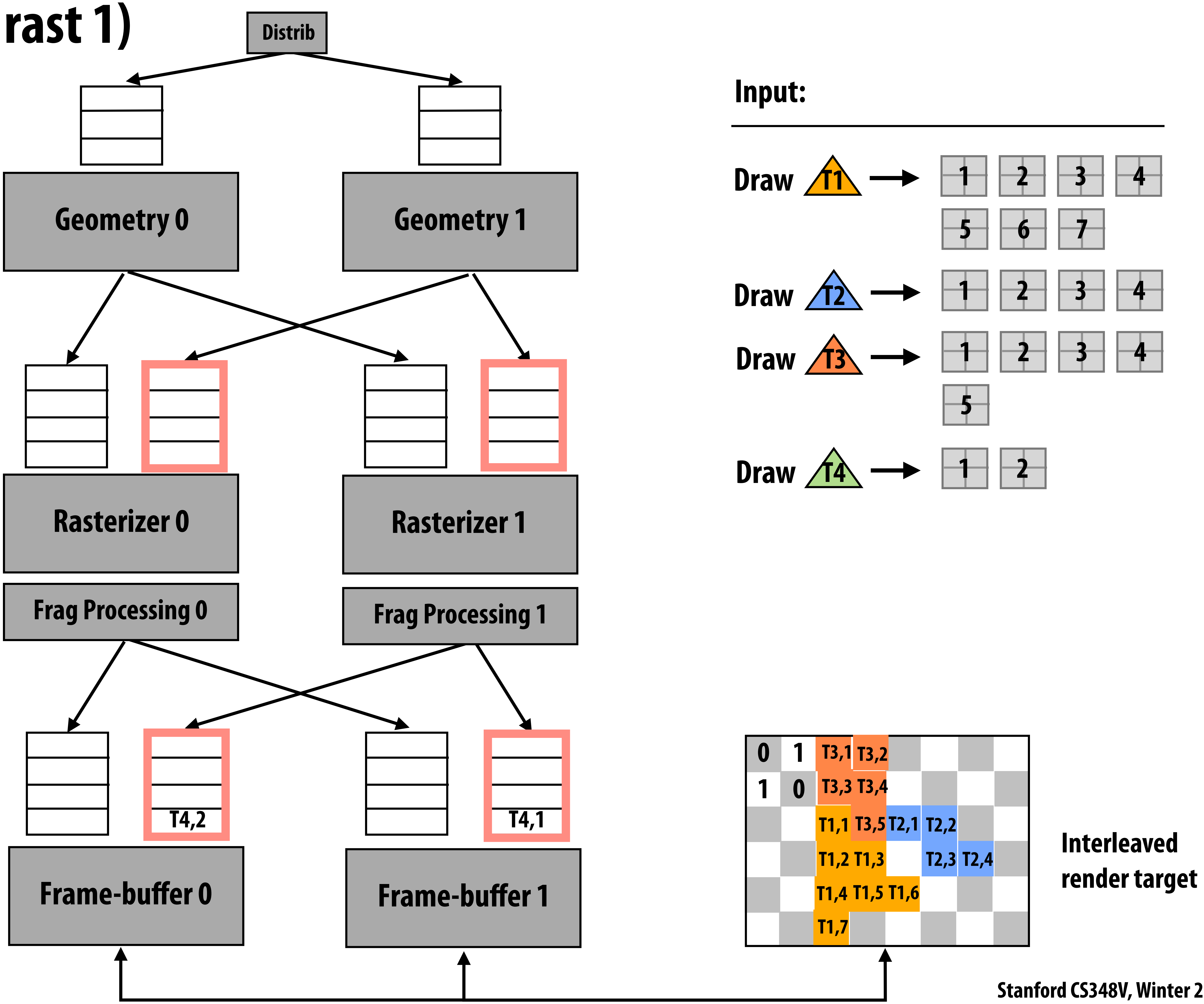
(Notice updates to frame buffer)



0	1	T3,1	T3,2				
1	0	T3,3	T3,4				
		T1,1	T3,5	T2,1	T2,2		
		T1,2	T1,3		T2,3	T2,4	
		T1,4	T1,5	T1,6			
		T1,7					

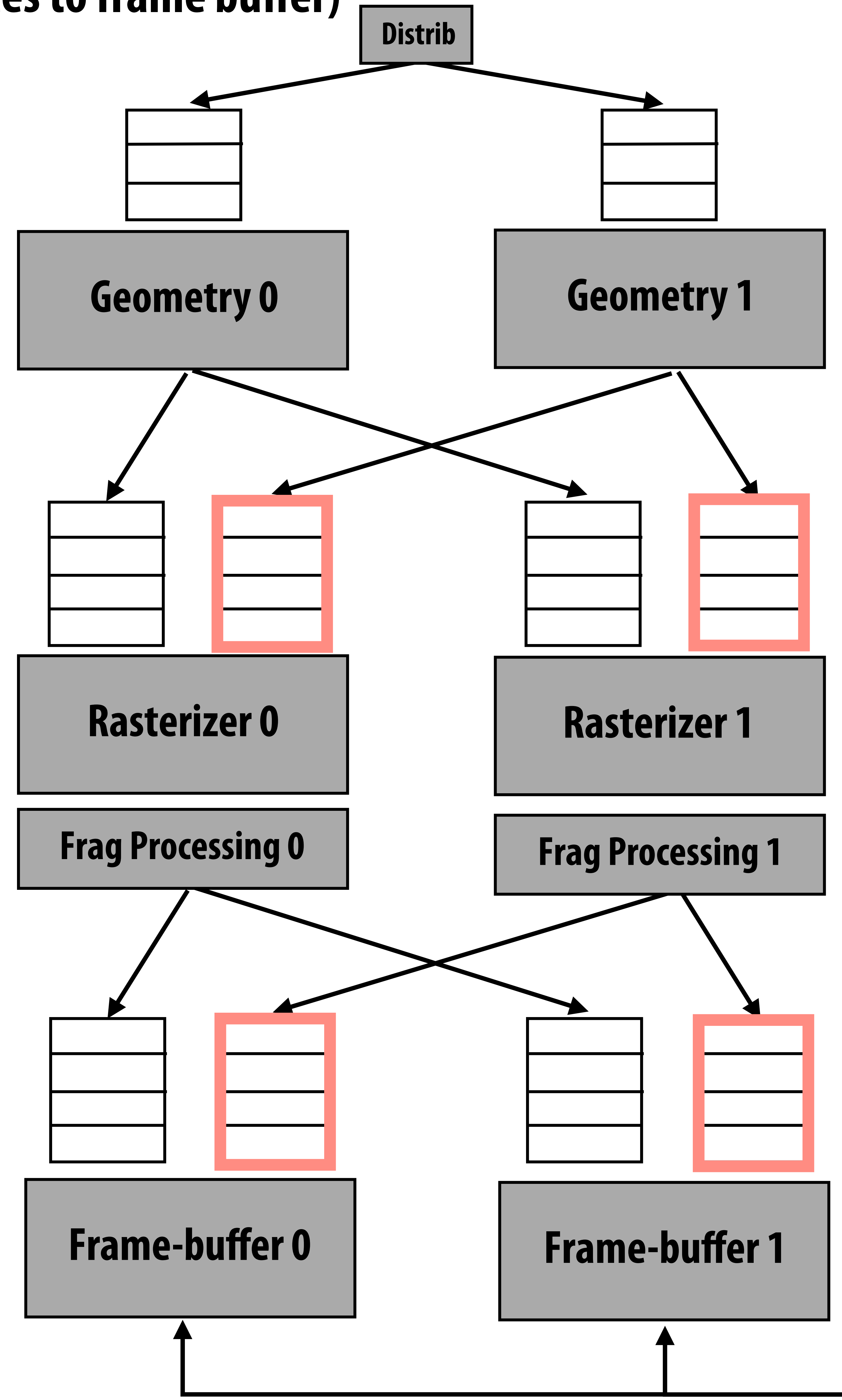
Interleaved render target

Switch token reached by FB: FB units start processing input from (geom 1, rast 1)



Frame-buffer units process frags from (geom 1, rast 1) in parallel

(Notice updates to frame buffer)



Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

Draw T4 → 1 2

0	1	T3,1	T3,2				
1	0	T3,3	T3,4			T4,1	T4,2
		T1,1	T3,5	T2,1	T2,2		
		T1,2	T1,3		T2,3	T2,4	
		T1,4	T1,5	T1,6			
		T1,7					

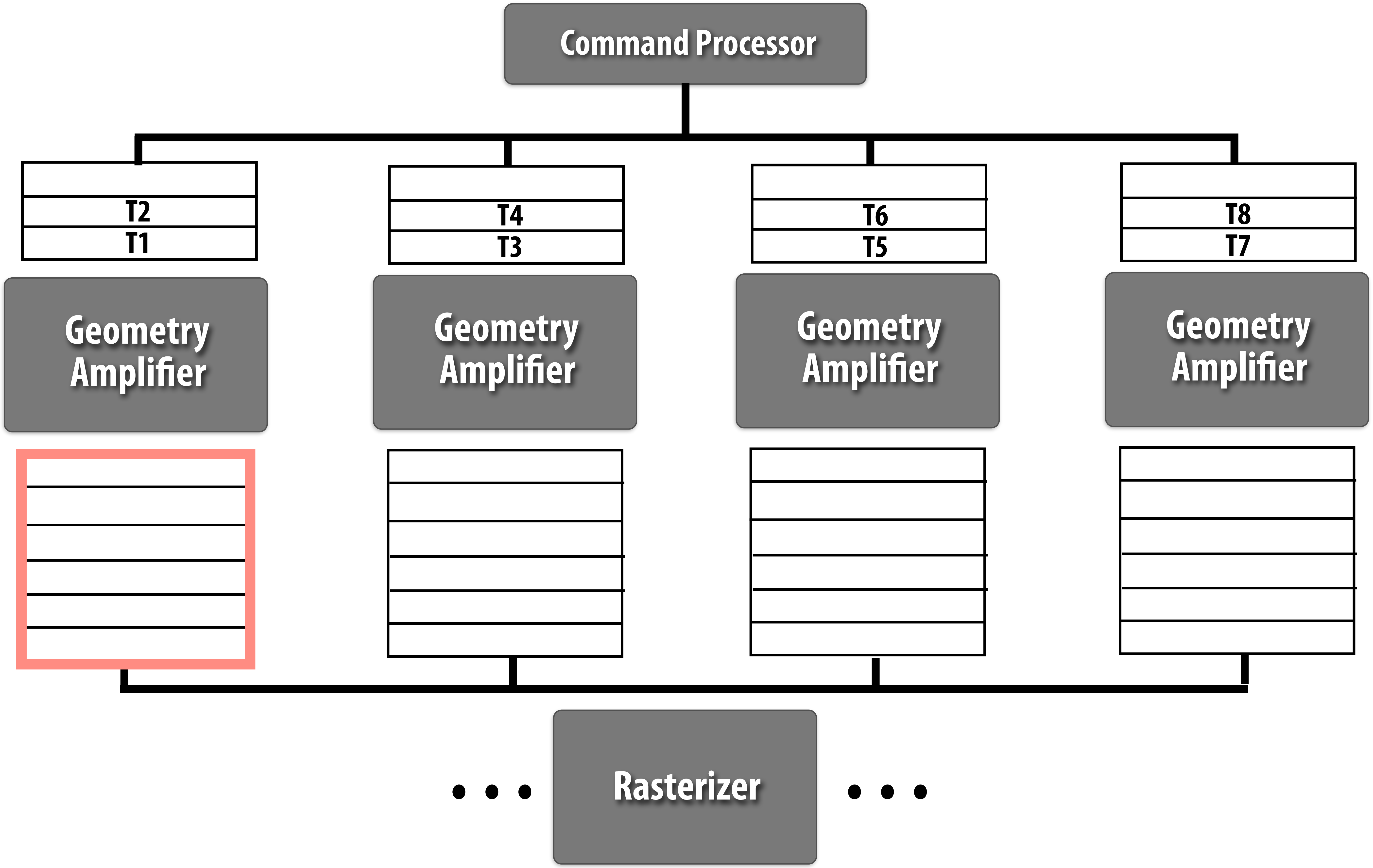
Interleaved render target

Parallel scheduling with data amplification

Geometry amplification

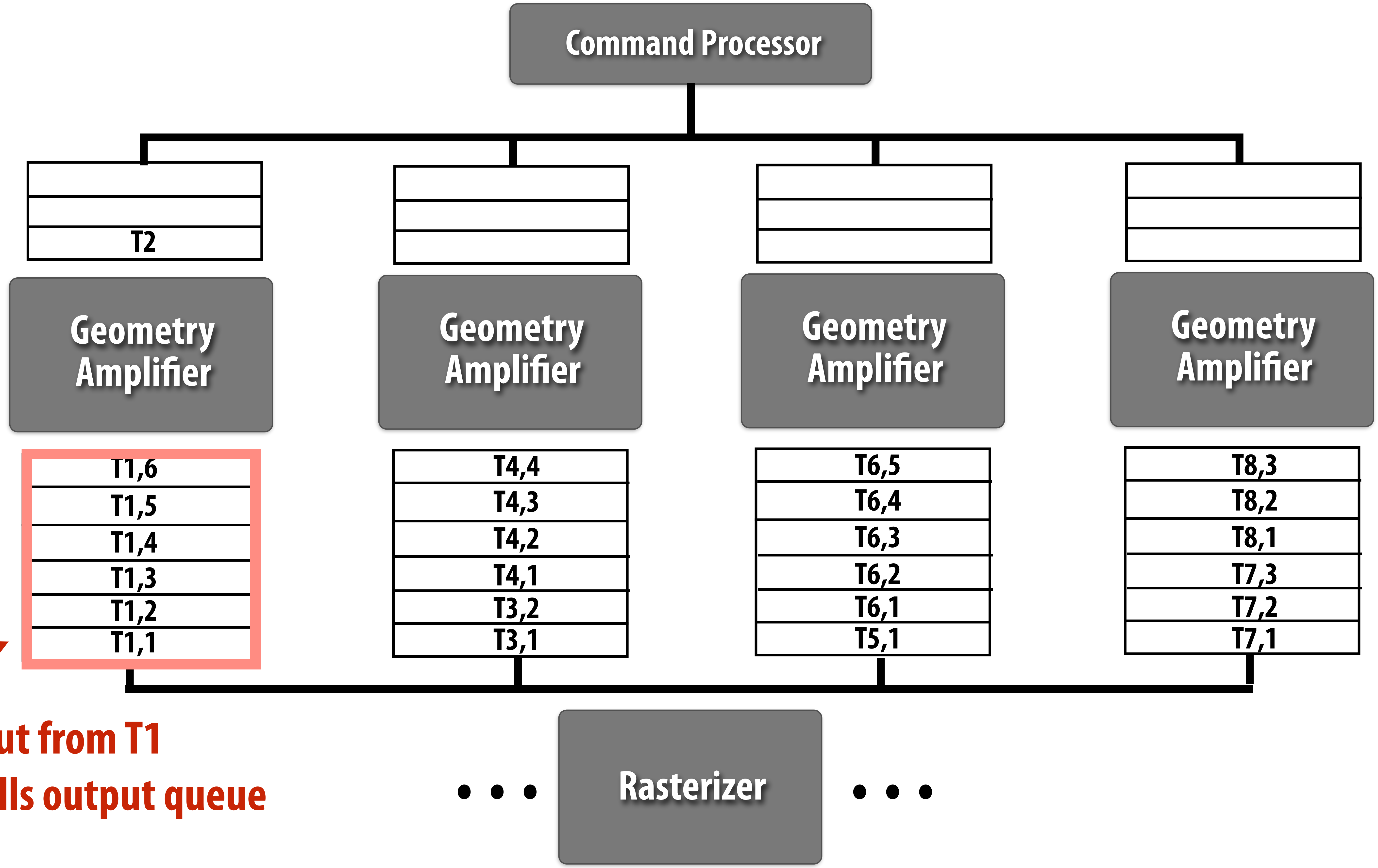
- Consider examples of one-to-many stage behavior during geometry processing in the graphics pipeline:
    - Clipping amplifies geometry (clipping can result in multiple output primitives)
    - Tessellation: pipeline permits thousands of vertices to be generated from a single base primitive (challenging 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

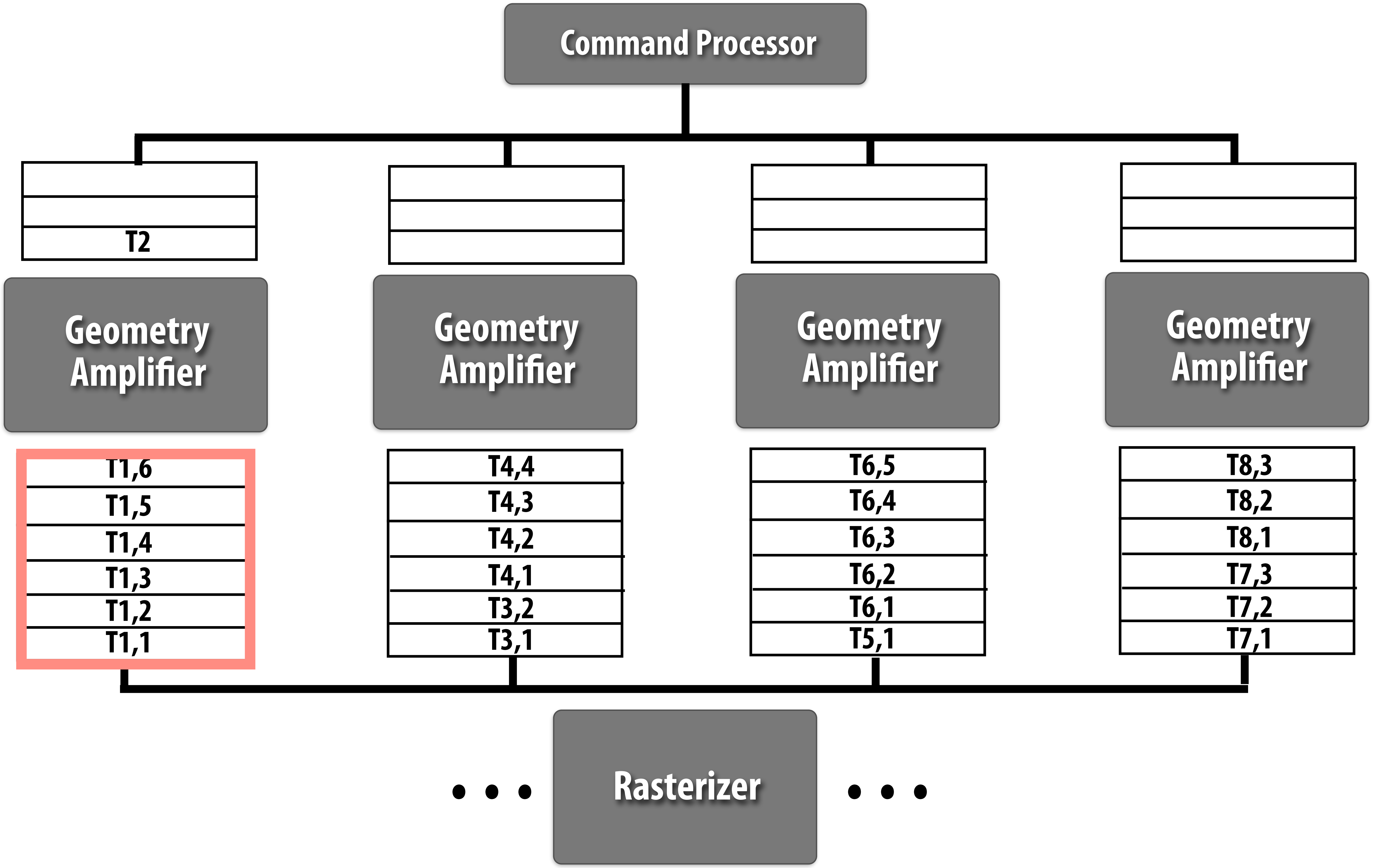


Notice: output from T1 processing fills output queue

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.

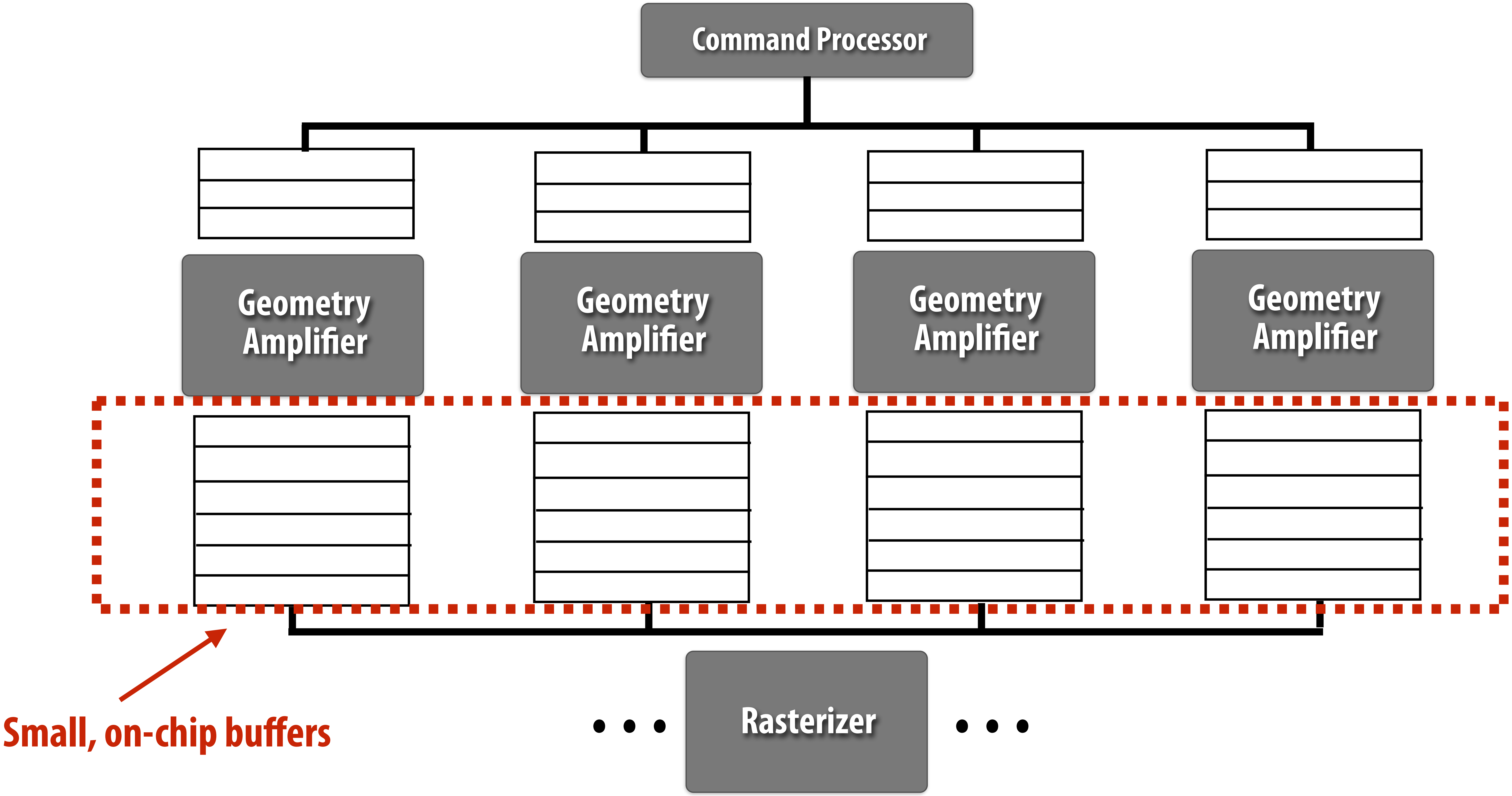
Thought experiment: design a scheduling strategy for this case



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

The <u>ideal solution</u> 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.

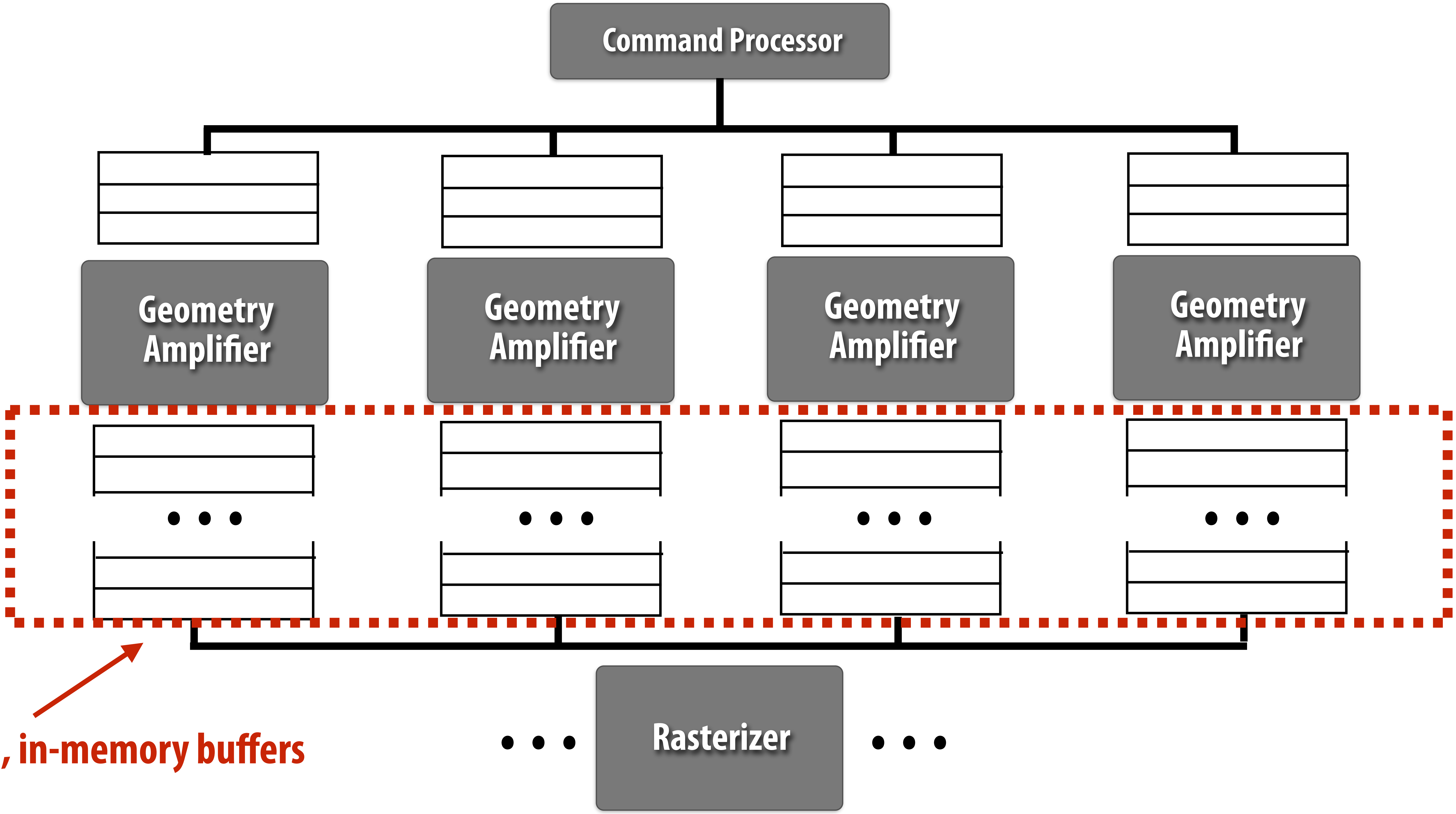
Implementation 1: fixed on-chip storage



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

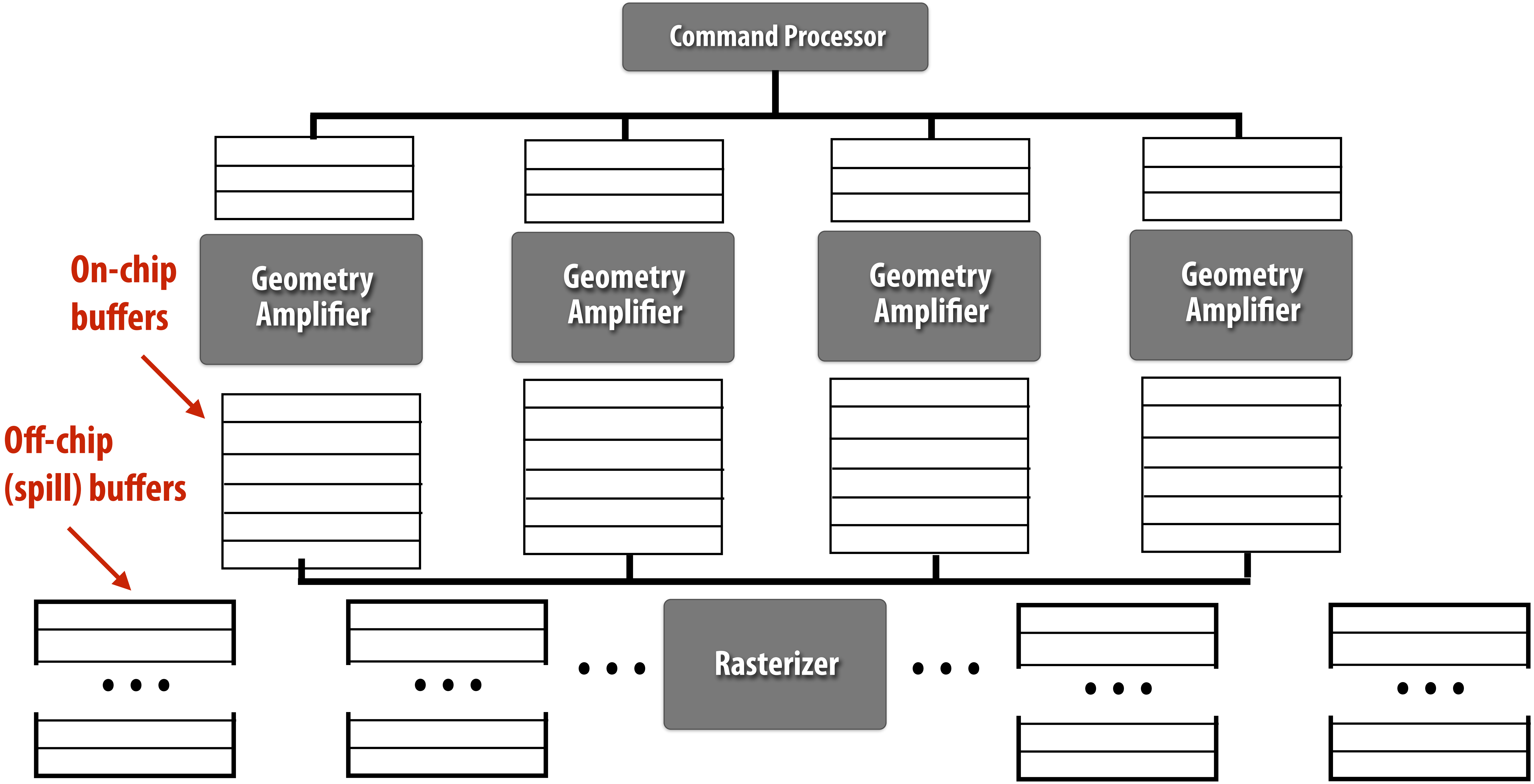


Large, in-memory buffers

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?

Implementation 3: hybrid

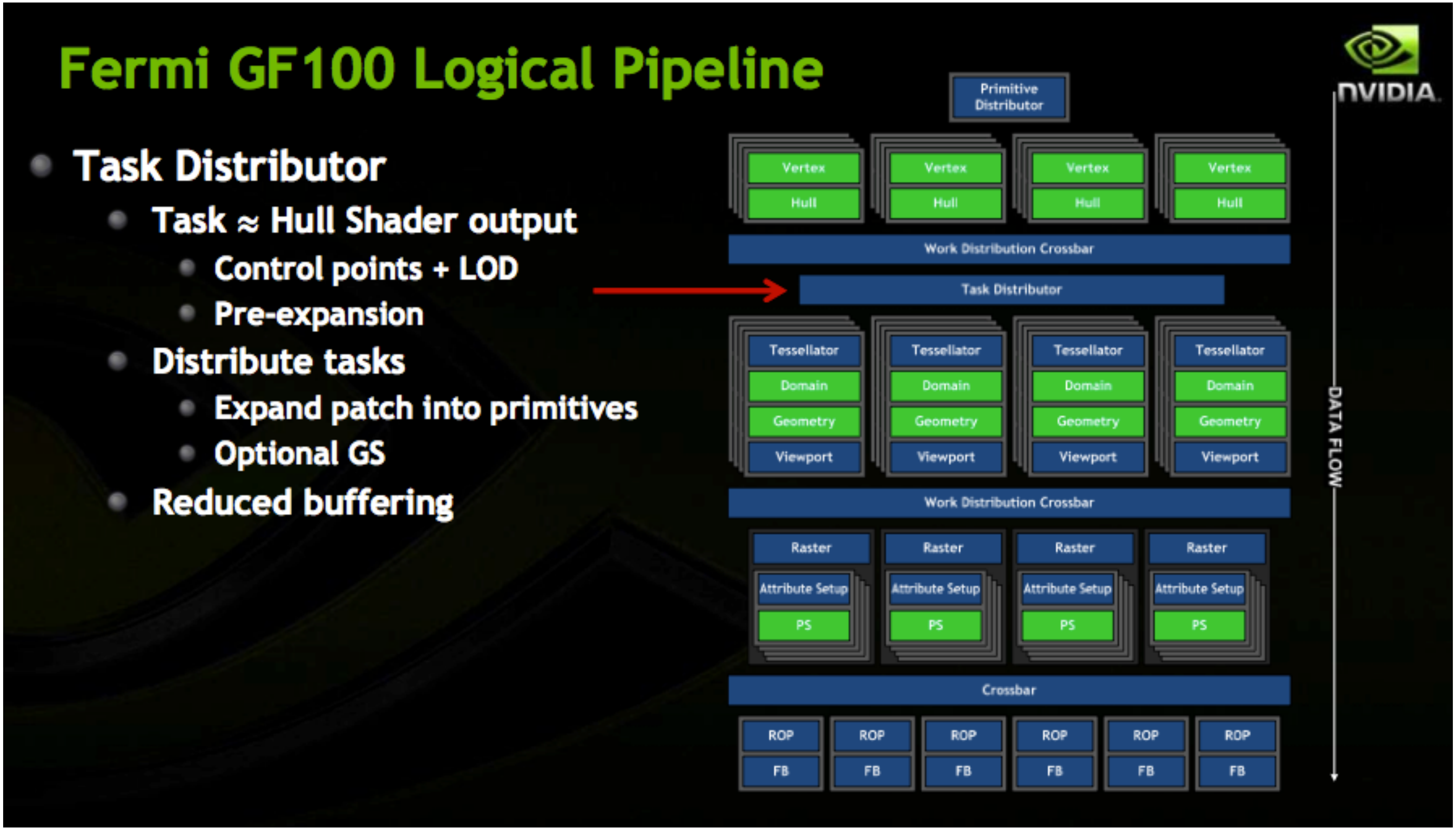


Hybrid approach: allocate output buffers on chip, but spill to off-chip, worst-case size buffers under high amplification

- **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 (since amplification factor known)



Fermi GF100 Logical Pipeline

- Task Distributor
  - Task ≈ Hull Shader output
    - Control points + LOD
    - Pre-expansion
  - Distribute tasks
    - Expand patch into primitives
    - Optional GS
  - Reduced buffering