Lecture 12:

Parallelism & Scheduling

EEC 277, Graphics Architecture
John Owens, UC Davis, Winter 2017

Logistics for Thursday

- Suggested 3 slides per group:
  - Motivation: why is your problem important?
  - Approach: how did you go about solving the problem?
  - Results: what did you learn?
  - Turn in slides in Canvas as PDF by 8a on Th morning
- 4 minutes per group
  - Historically people go long—practice!
  - Two-person groups: both must speak
- If you have a demo, a) let me know and b) put your slides on the same machine as your demo

Tips on Presentations

- Most important: Impact on graphics systems
- Consider your audience
  - Likely that your colleagues don't know your subject in much detail
  - I don't either
- Thus, concentrate on high level and results
- Pick a few interesting points and explain them well

Your Writeups

- Talk: Graded on effectiveness of talk
- Writeup: Graded on content
- Most important: Impact on graphics systems
  - What did you learn about graphics systems?
  - How will what you learned impact future graphics systems?

Your Writeups

- Consider …
  - Impact on hardware design
  - Impact on API/programming system?
  - Impact on overall performance?
  - Impact on runtime (how much?), quality (how good?), bandwidth (how much?)
- What is efficient (uses the hardware well) and what is inefficient? How would you propose changing what is inefficient?
- Generality?
- Scalability?

Your Writeups

- Think about presenting to our guest speakers …
  - High-level motivation: Why is the problem you chose important?
  - Technical detail: Why is your solution a good one? (Make sure to detail your contribution!)
  - Describe previous work
  - Concentrate on impact on graphics systems
  - Throughout, think about: Analysis!
- http://www.ece.ucdavis.edu/~jowens/commonerrors.html
- http://www.ece.ucdavis.edu/~jowens/biberrors.html

Graphics Has Parallelism

- Why?
  - Faster
  - Bigger
  - “Time machine”

- 100k–1M+ triangles/frame, parallelizable
- 1–10M+ pixels/frame, parallelizable
  - And fragments/samples have even more parallelism
- So what’s the problem?

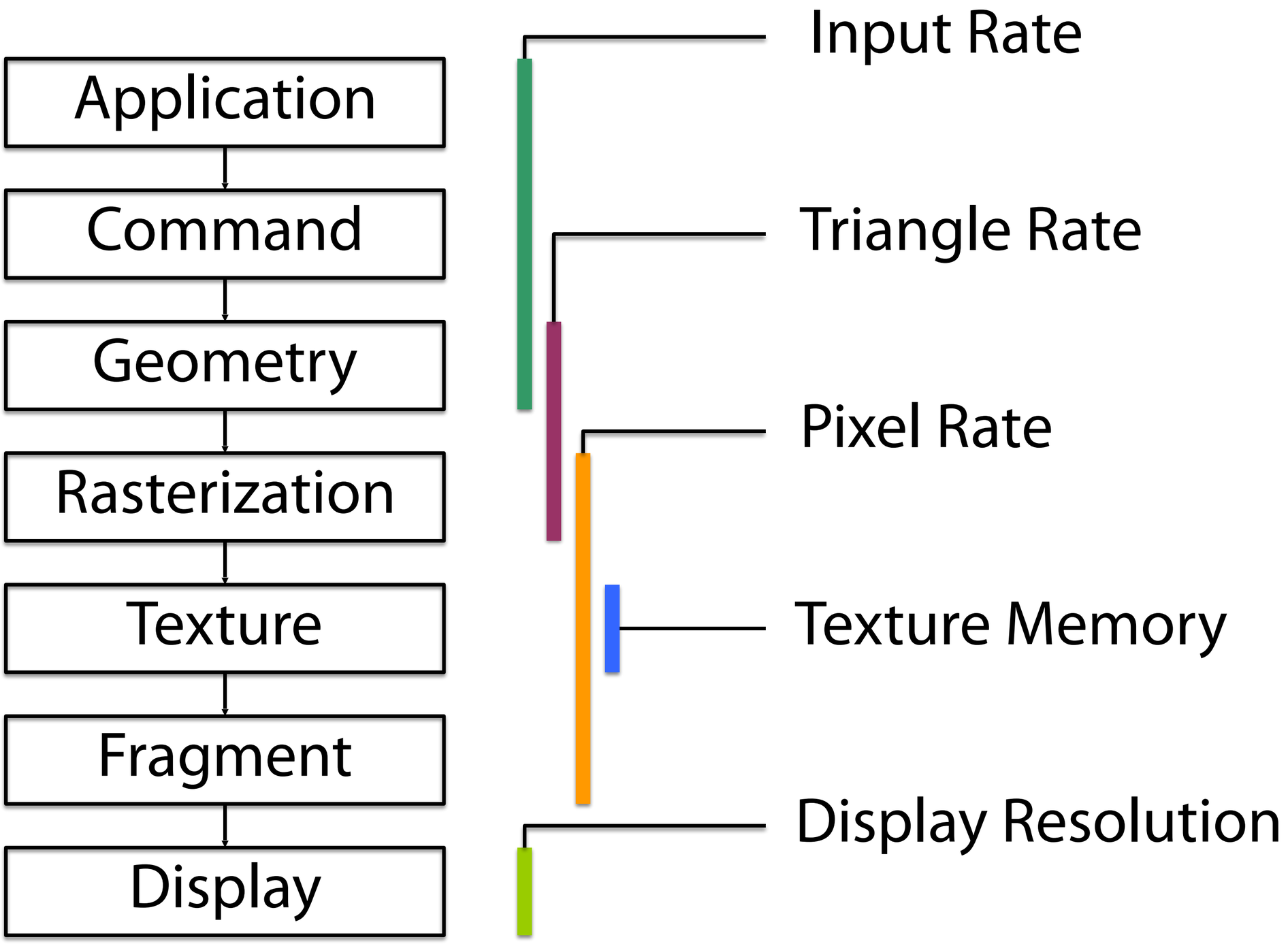
Difficulties with Parallelization

- Ordering
  - Graphics state
  - Framebuffer ops (painter's algorithm)
  - Render to texture / Readback
- Load-balancing
  - Varying primitive work +
  - Varying primitive position =
  - Idle processors and work duplication
- Host bandwidth

Outline

- Terminology and Strategies
- Communication Structures / Costs
- Taxonomy
- Scheduling

Measuring Performance



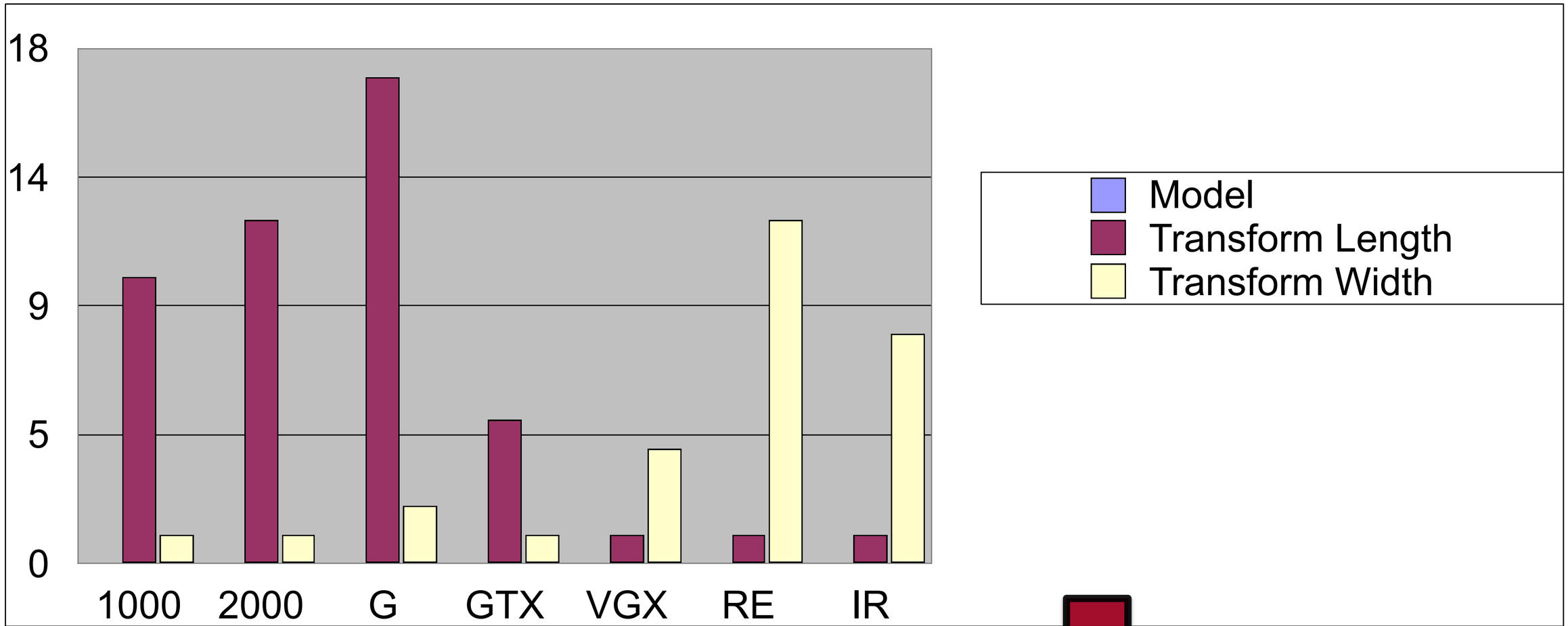
Scalable Graphics Means …

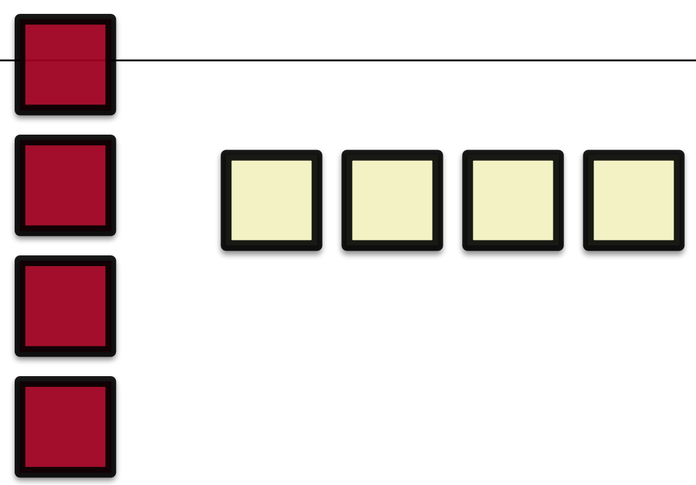
- Increase all performance metrics
- *n* pipelines results in …
  - Same input *n* times faster
  - *n* times bigger input in same time
- $O(n)$ throughput
- $O(1/n)$ latency
- $O(n^2)$ cost is bad!

Sources of Parallelism

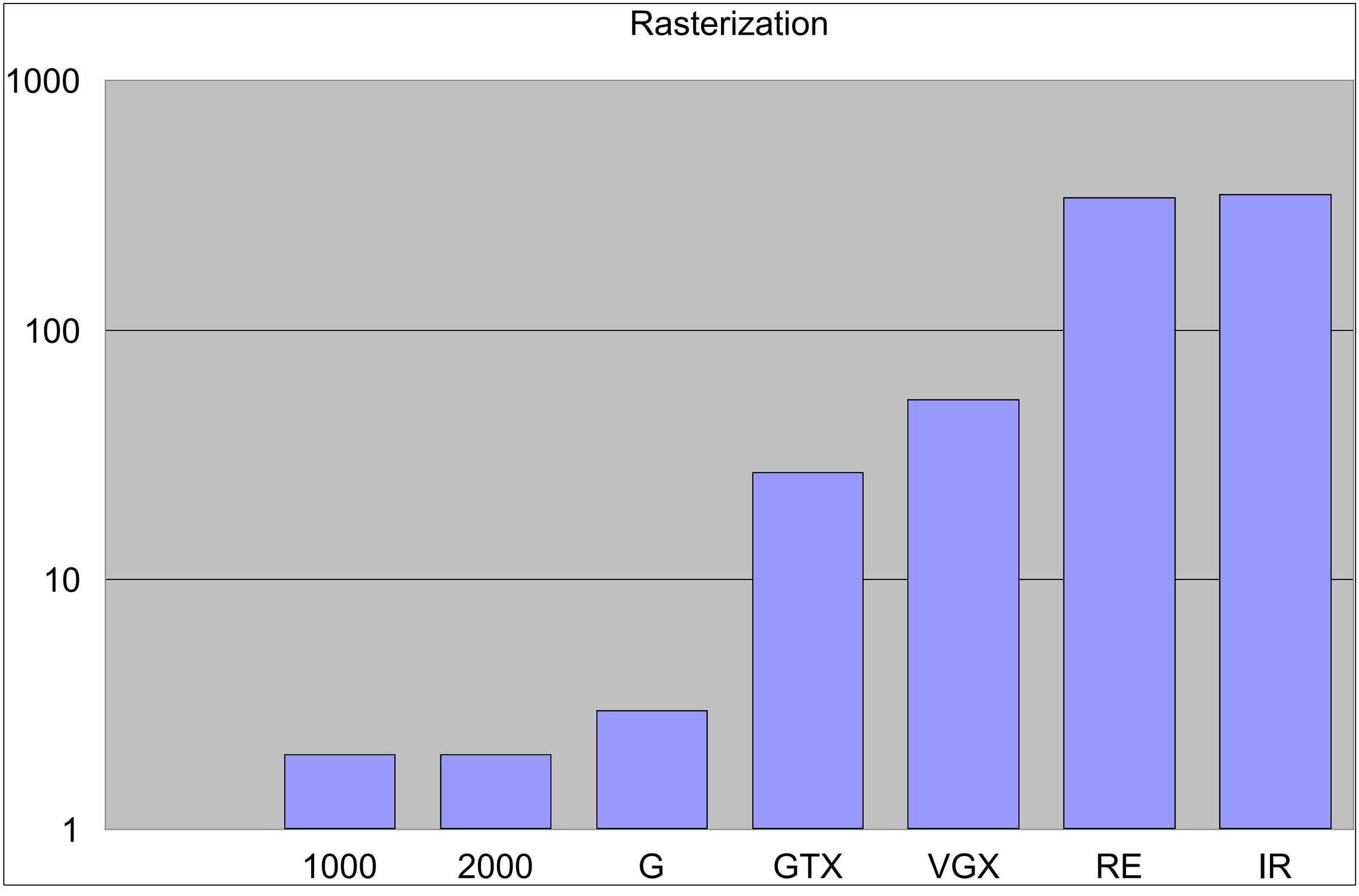
- Task parallelism
  - Graphics pipeline—between multiple stages
- Data parallelism
  - Frame-parallel
  - Image-parallel
  - Object (Geometry)-parallel
- Interframe vs. Intraframe

Geometry Parallelism (SGI)



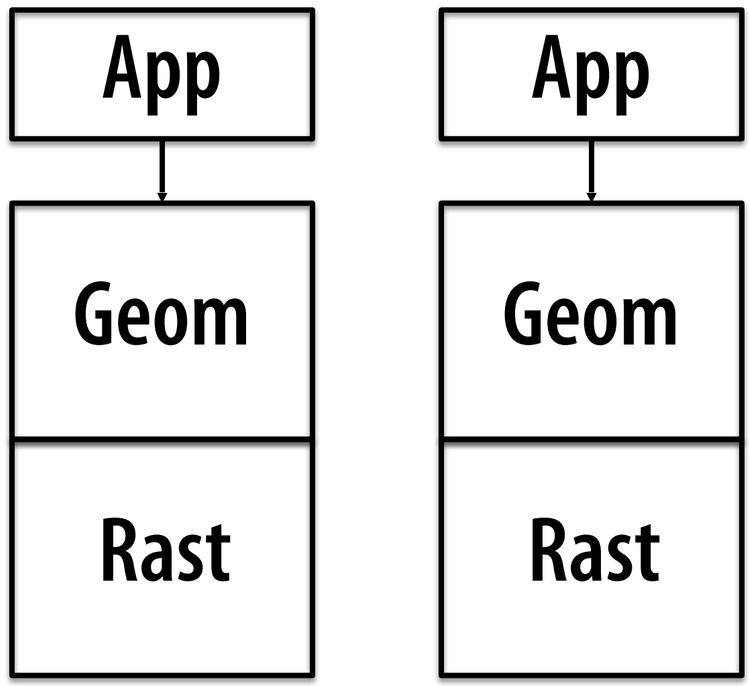


Raster/Fragment Processors (SGI)



Sorting as a Taxonomy

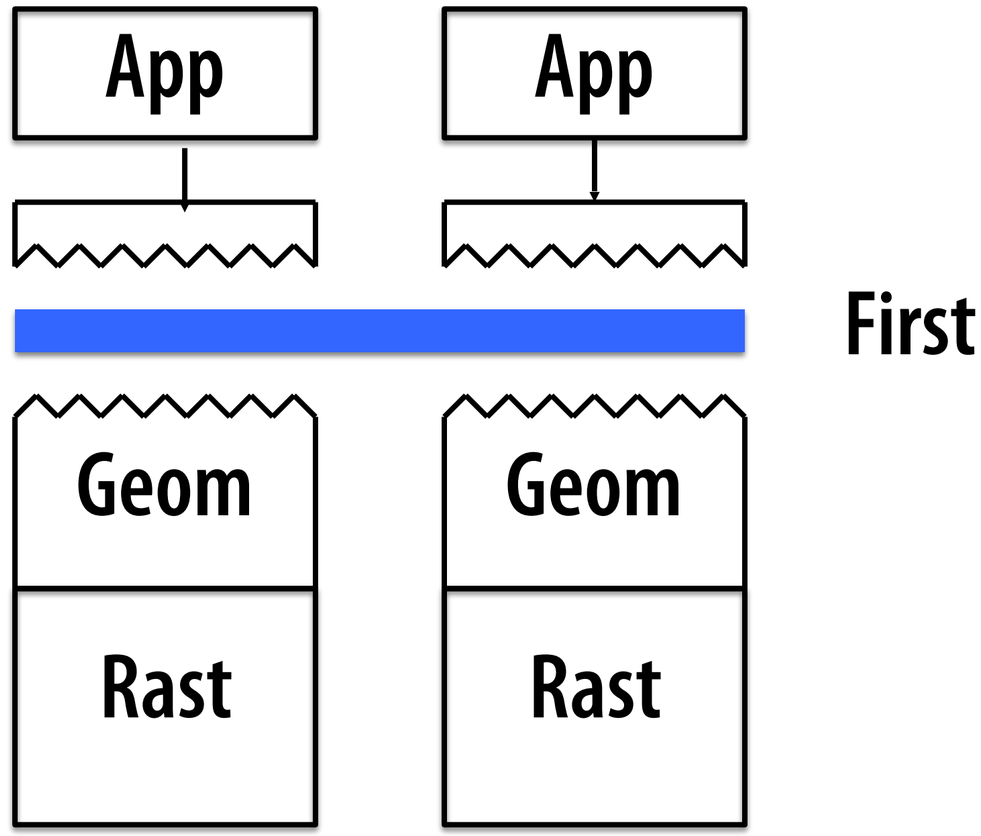
- Object parallel input
- Image parallel output
- "Sort" between them
  - Communicate work to the correct location on the screen



A Sorting Classification of Parallel Rendering, Molnar et al.

Sorting as a Taxonomy

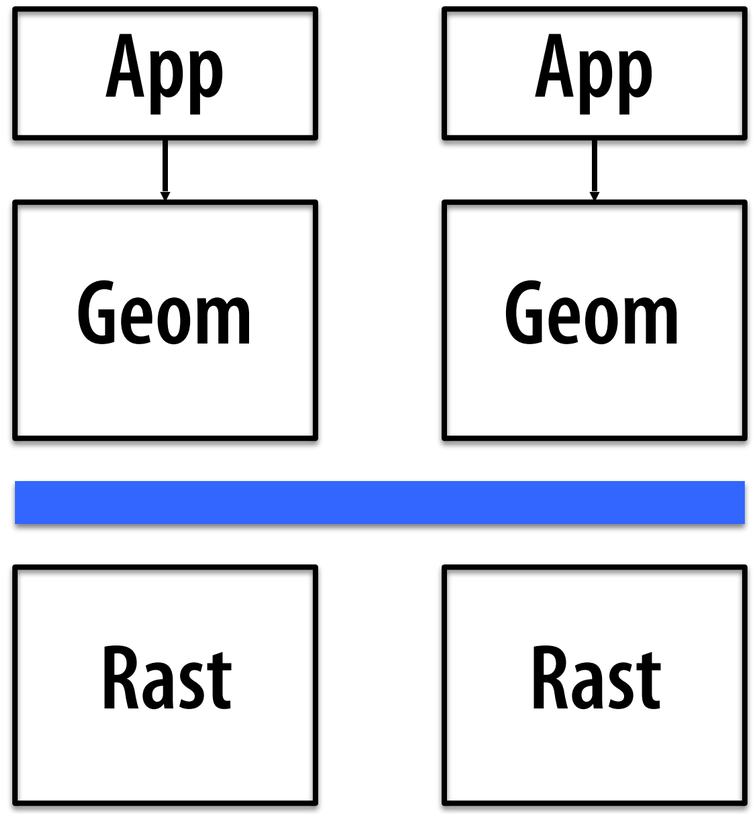
- Object parallel input
- Image parallel output
- "Sort" between them
  - Communicate work to the correct location on the screen



A Sorting Classification of Parallel Rendering, Molnar et al.

Sorting as a Taxonomy

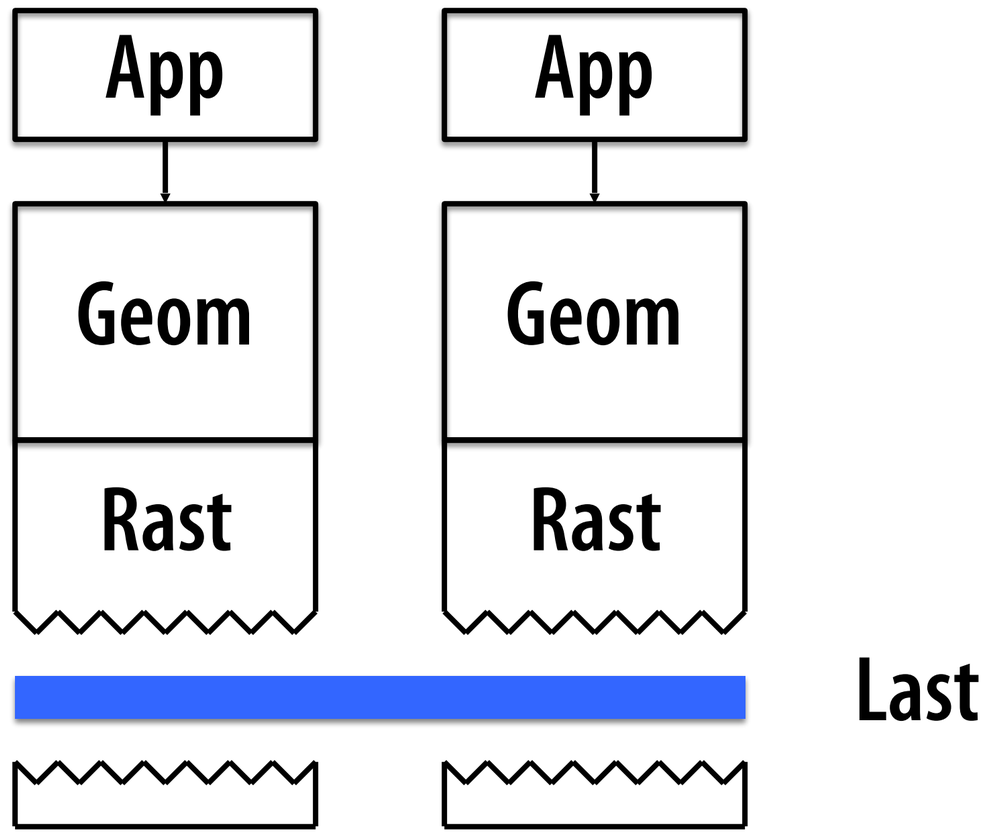
- Object parallel input
- Image parallel output
- "Sort" between them
  - Communicate work to the correct location on the screen



A Sorting Classification of Parallel Rendering, Molnar et al.

Sorting as a Taxonomy

- Object parallel input
- Image parallel output
- "Sort" between them
  - Communicate work to the correct location on the screen



A Sorting Classification of Parallel Rendering, Molnar et al.

Object Parallelism

- Distribute geometry work among processors
- Challenge: Load imbalance
  - Completely controlled by work distribution
  - Difficult to estimate cost of work
  - If work is submitted in parallel, how to coordinate distribution?
    - Each parallel unit attempts balance—OK?
- Challenge: State management
- Challenge: Enforcing order at end
  - Usually point of synchronization

RealityEngine Geometry (1992)

- Variable-functionality MIMD organization
  - Eight identical engines
  - Round-robin work assignment
  - Good 'static' load balancing
- Command processor
  - Splits large strips of primitives
  - Shadows per-vertex state
  - Broadcasts other state
- Primitive assembly
  - Complicates work distribution
  - Reduces efficiency of strip processing
- Reordering at end?

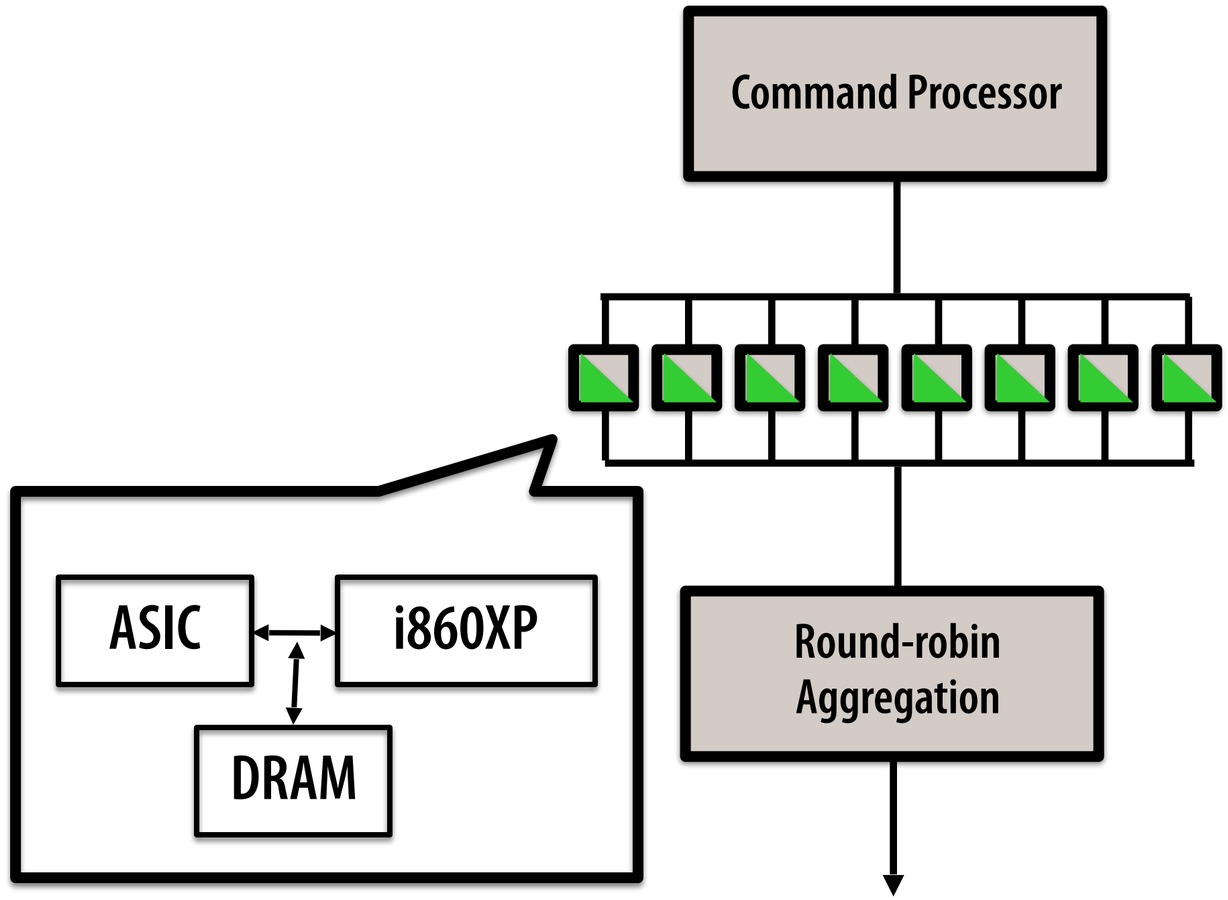
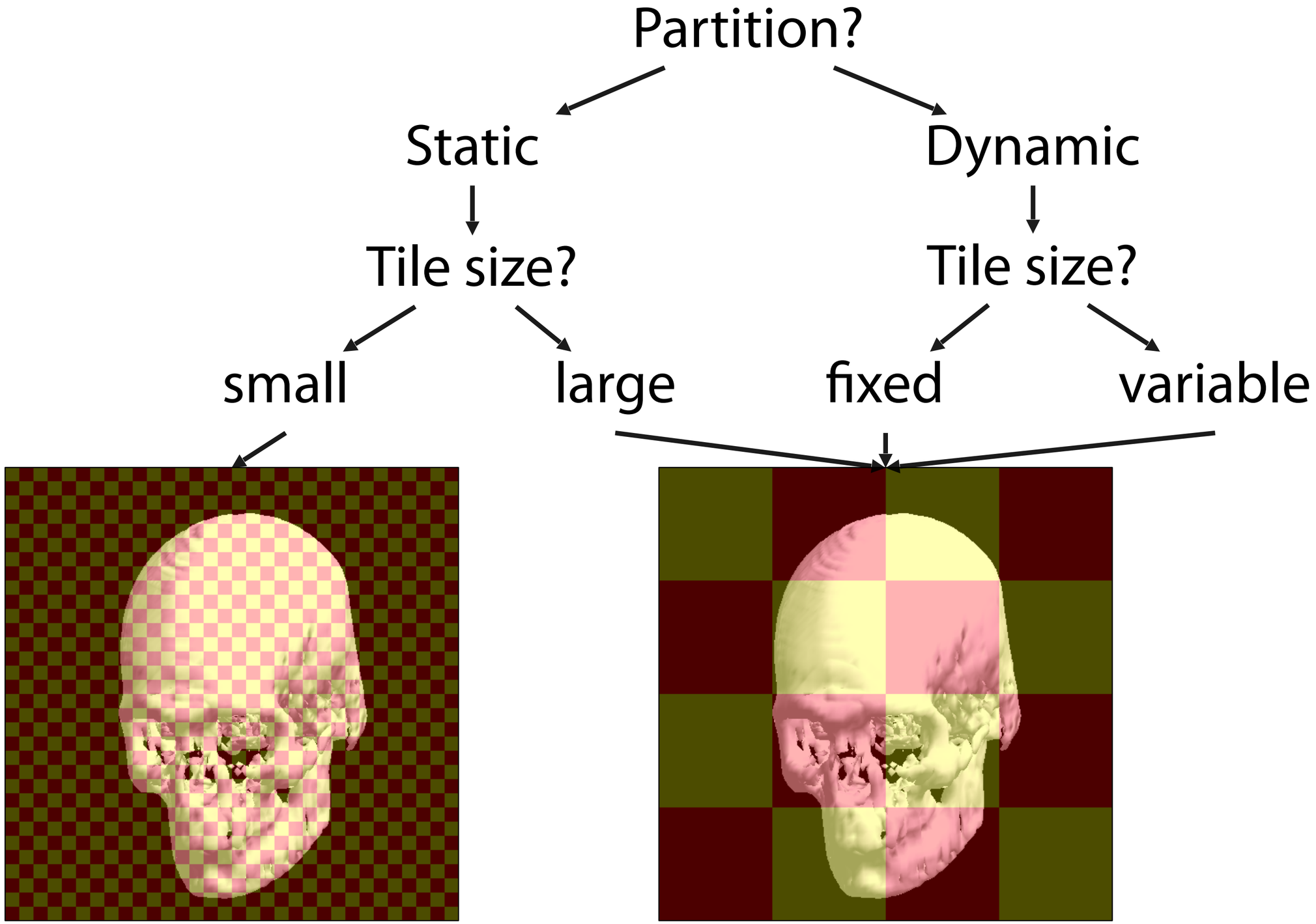


Image Parallelism

- Divide screen into regions, one or many regions per processor
- Must know (approx.) where work is on screen
  - Possible overlap
- Considerations
  - Dynamic or static partition?
  - Size of partition?

Classification of Tiling



Fuchs - Interleaved

Parke - Tiled

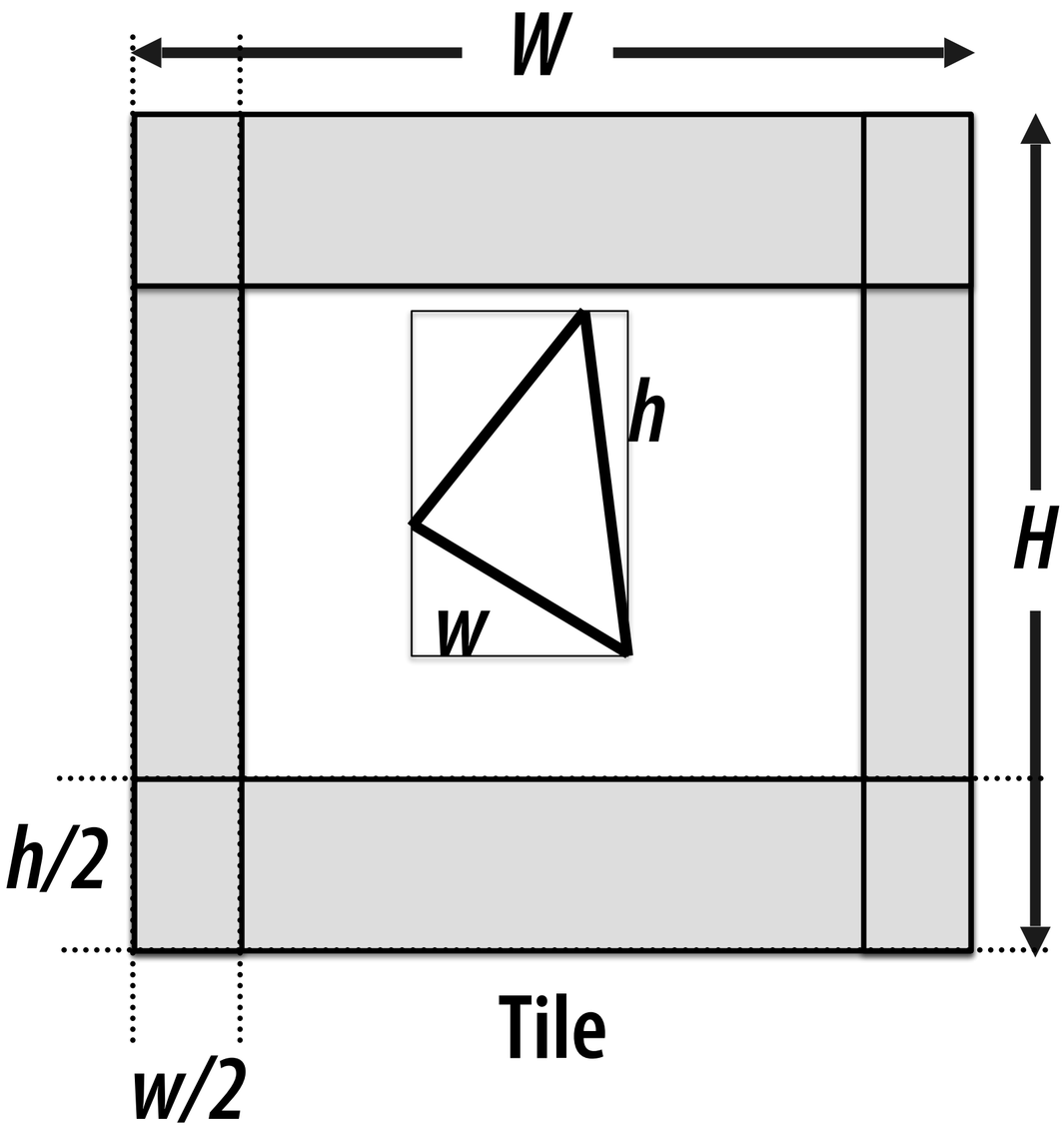
Image Space Decomposition Choices

- Big tiles vs. small tiles (interleave):

- Small tiles:
  - Better load balance, but overlap
  - Requires broadcast

- Big tiles:
  - Non-uniform screen space coverage → load imbalance
  - Temporal effects → load imbalance

The Overlap Factor



Molnar-Eyles Formula

3 cases

$$4 \times \frac{4(w/2)(h/2)}{WH}$$

$$2 \times \frac{2(w/2)(H-h) + 2(h/2)(W-w)}{WH}$$

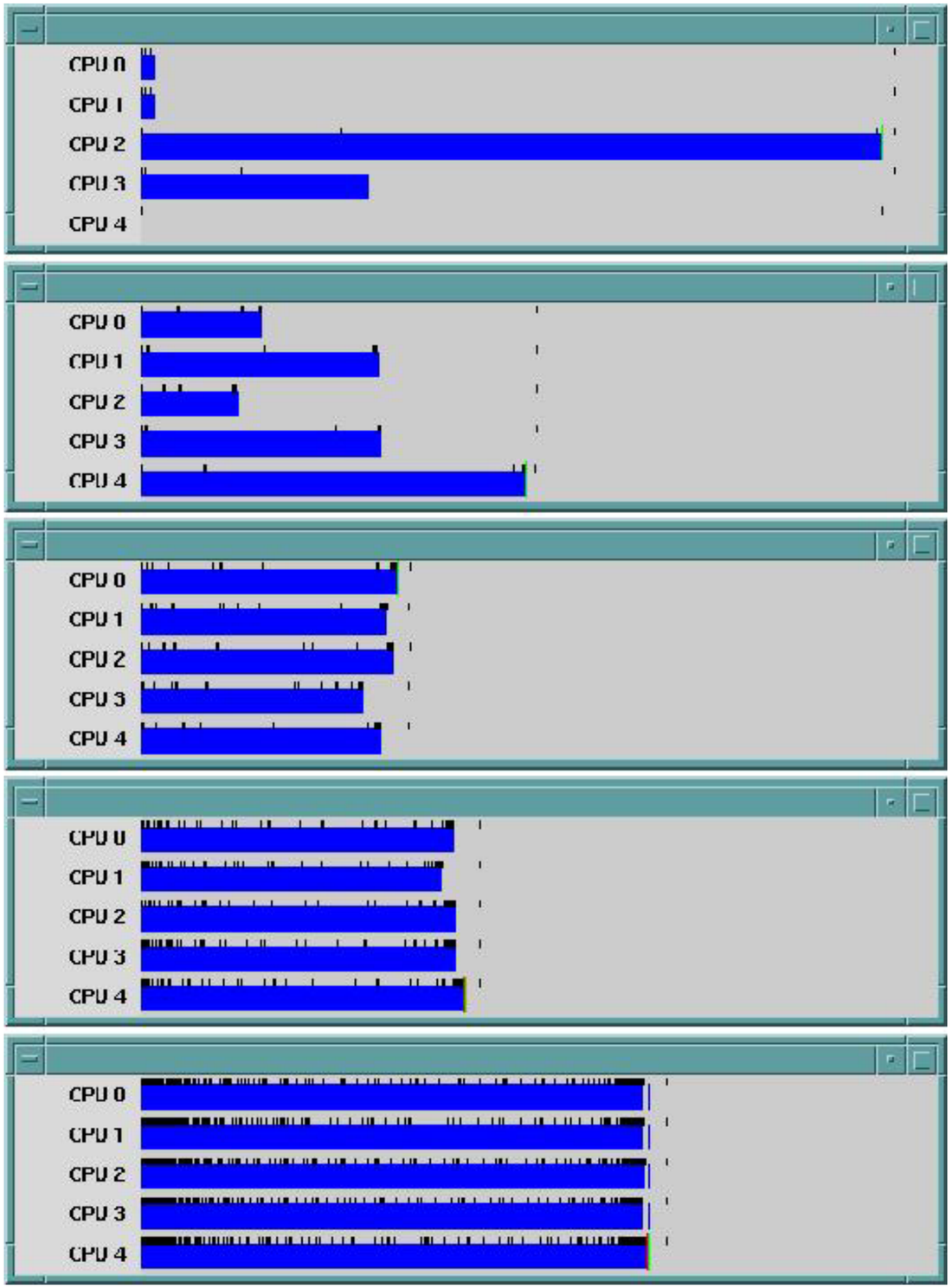
$$1 \times \frac{(W-w)(H-h)}{WH}$$

Total

$$O = \left(\frac{H+h}{H}\right)\left(\frac{W+w}{W}\right) = \left(1+\frac{h}{H}\right)\left(1+\frac{w}{W}\right)$$

Rasterization Cost

- Large tiles
  - Few tasks, greater variation in work
  - → bad load balance
- Medium tiles
  - More tasks, low overlap
  - → good load balance
- Small tiles
  - High overlap/more redundancy
  - → best load balance but redundant work



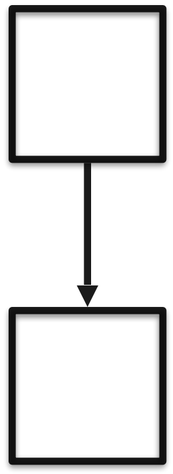
Why Communicate?

- In parallel graphics, two reasons:
  - Allows fan-in and fan-out
  - Increases efficiency
    - Load-balancing
    - Reduces duplication of work and storage

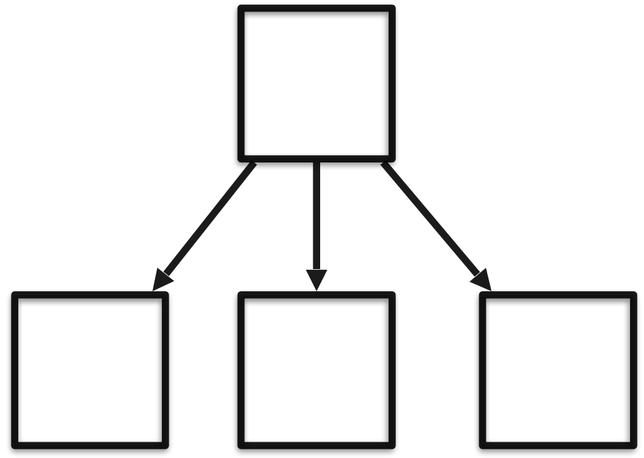
Outline

- Terminology and Strategies
- Communication Structures / Costs
- Taxonomy
- Scheduling

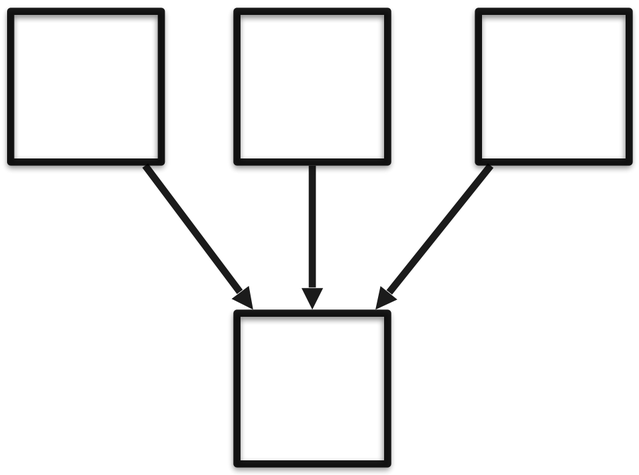
Types of Communication



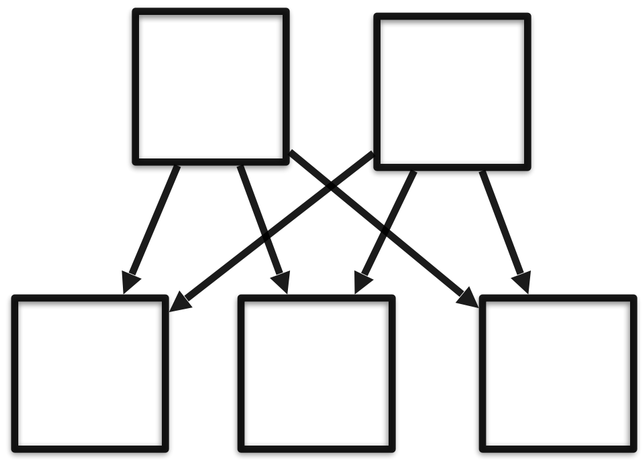
1 to 1



1 to N (Broadcast)



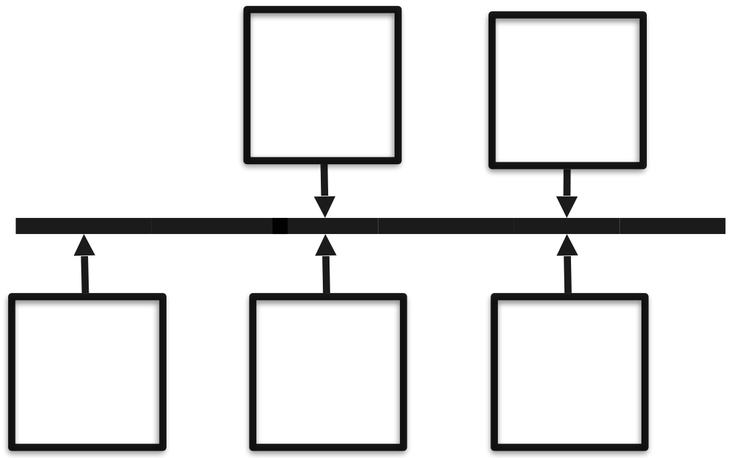
N to 1



N to M

Types of Communication: Bus

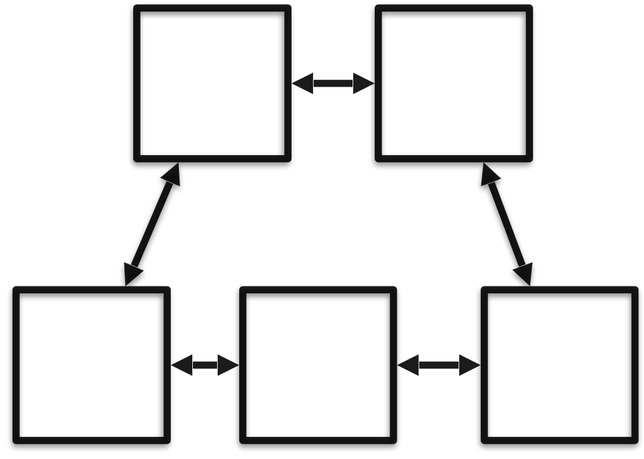
- $O(n)$ per-port cost, $O(n^2)$ total
  - Not particularly scalable
    - Each server gets bus 1/nth of time
    - Bus width should be $O(n)$
    - Electrically difficult to scale—speed decreases with more sources/destinations



- Broadcast same cost as 1-to-1
- Can provide serialization
- In practice: target maximum degree of parallelism
  - Non-scalable bus
  - When bus not fully used, wasted cost

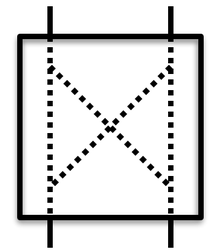
Types of Communication: Ring

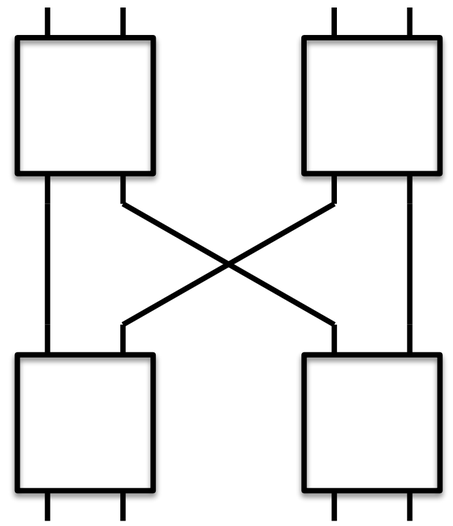
- Interconnects multiple units with series of 1-to-1 neighbor communications
- *O*(1) per-port cost, *O*(*n*) total
  - Good scaling characteristic
  - Well suited for broadcast
  - Great for nearest-neighbor



Types of Communication: MIN

- Intermediate switch elements
  - Butterfly, omega, fat trees …
  - Point-to-point communication
- *O*(*n* log *n*) total cost

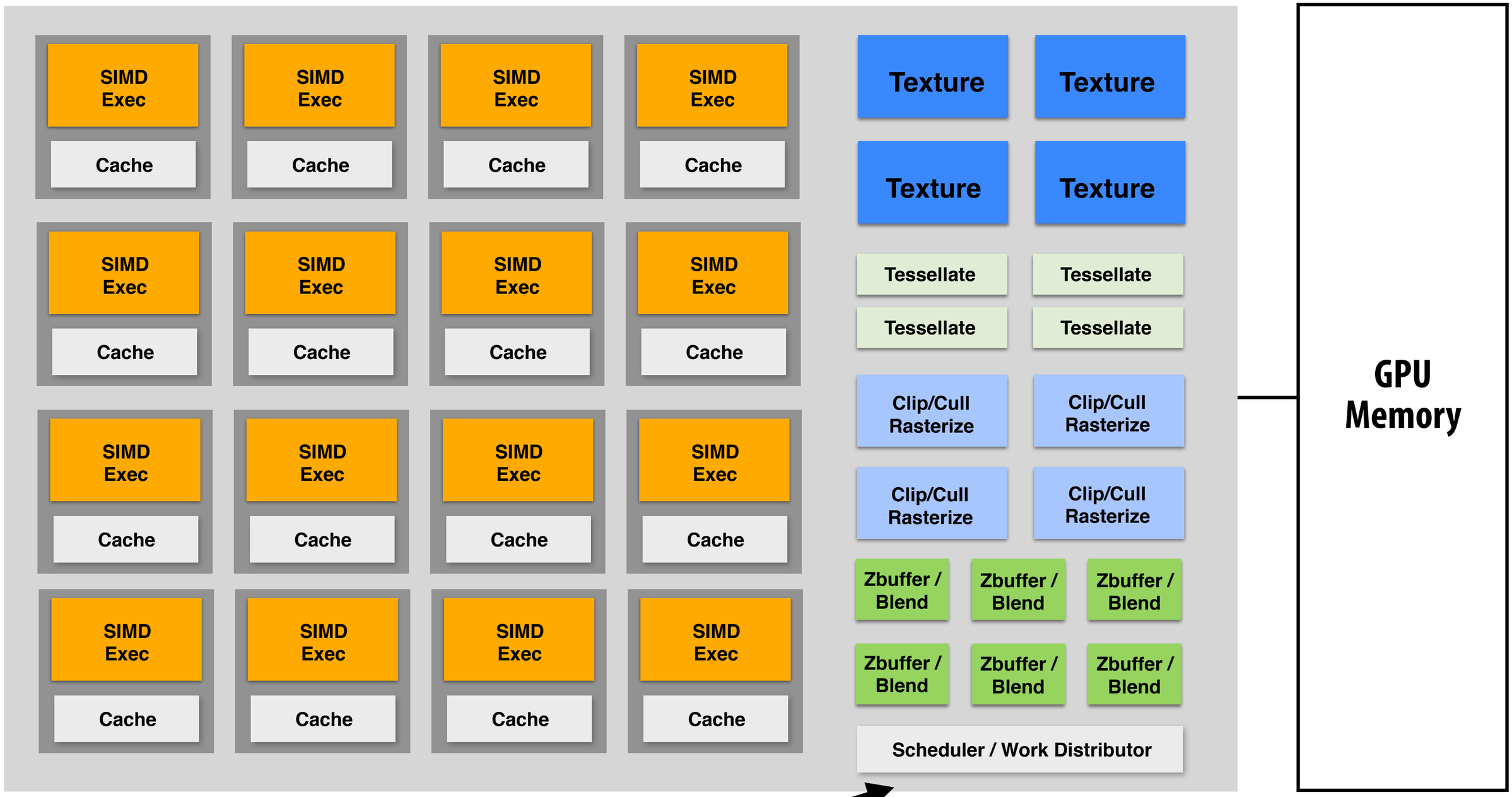




Outline

- Terminology and Strategies
- Communication Structures / Costs
- Taxonomy
- Scheduling

GPU: heterogeneous parallel processor



We're now going to talk about this scheduler

Reminder: requirements + workload challenges

- Immediate mode interface: pipeline accepts sequence of commands
  - Draw commands
  - State modification commands

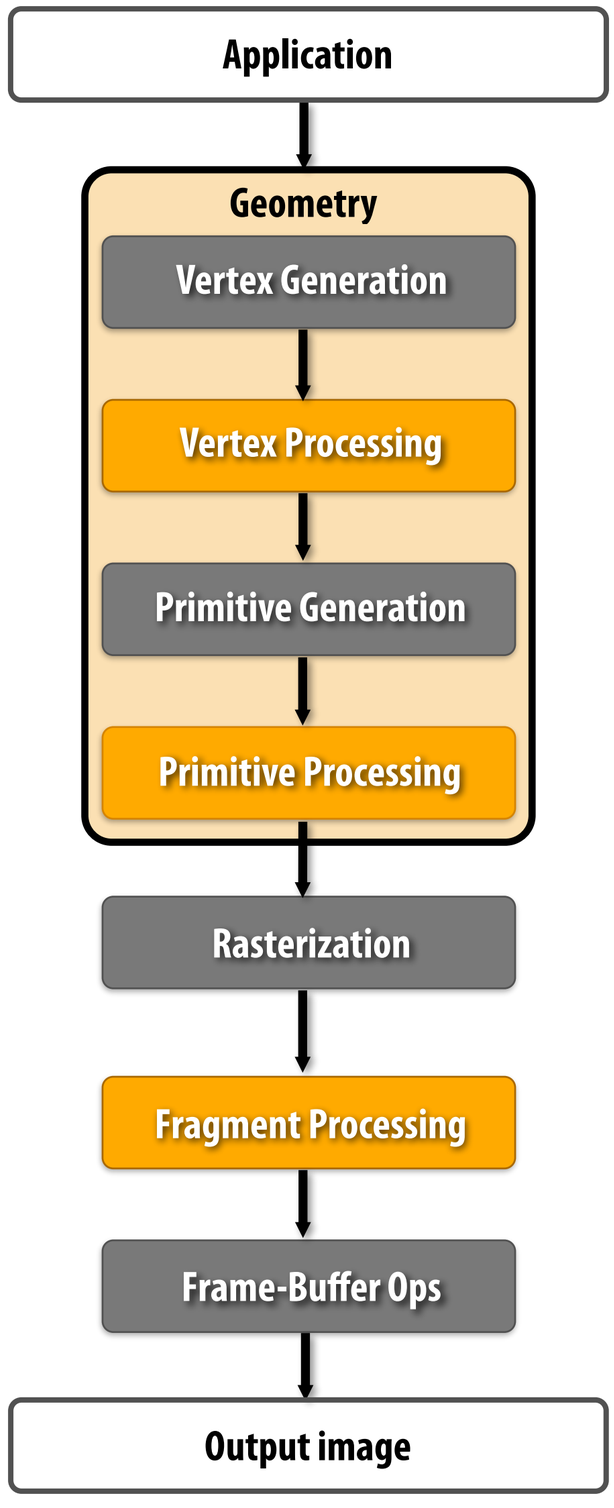
- Processing commands has sequential semantics
  - Effects of command A must be visible before those of command B

- 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.

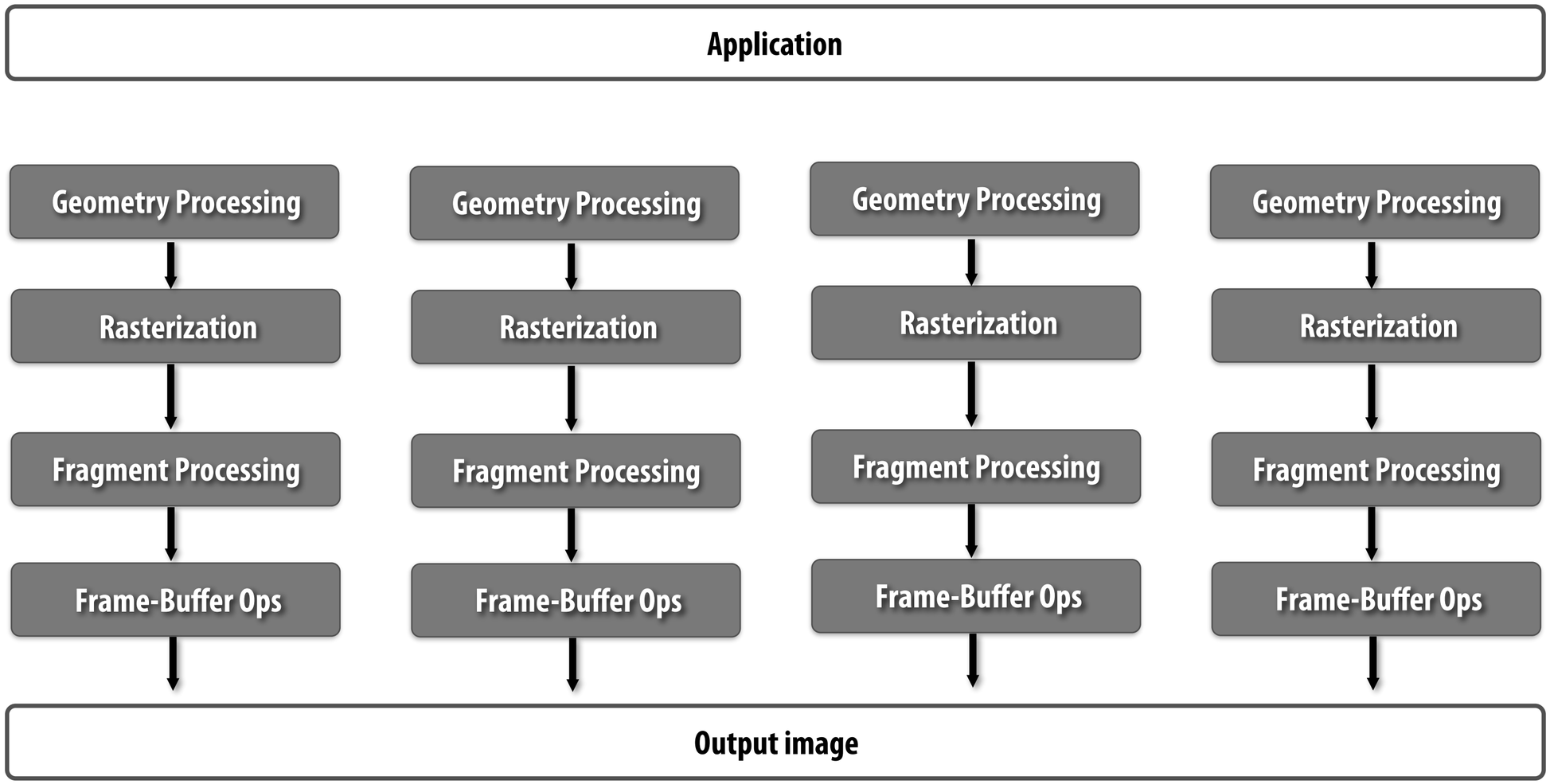
Simplified pipeline

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



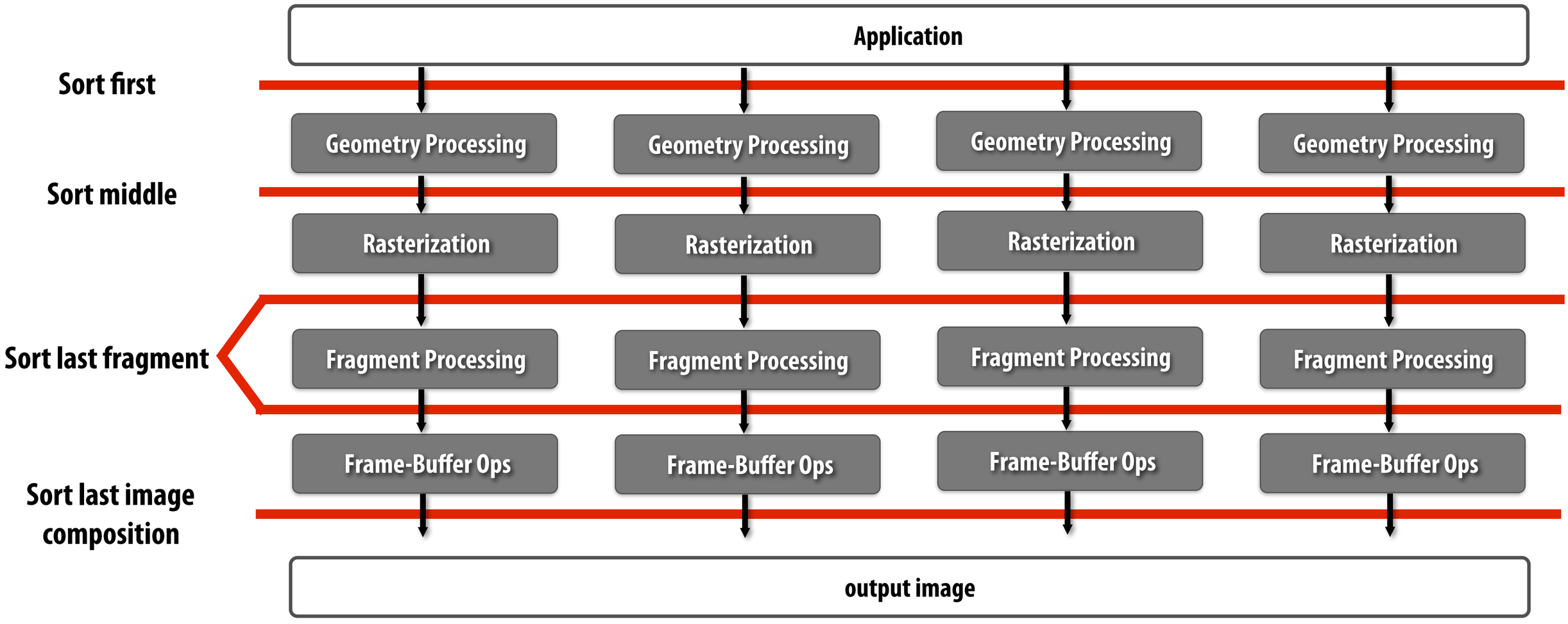
A cartoon GPU:

- Assume we have four separate processing pipelines
- Leverages data-parallelism present in rendering computation



Molnar & Eldridge's sorting taxonomy

- Implementations characterized by where communication occurs in pipeline

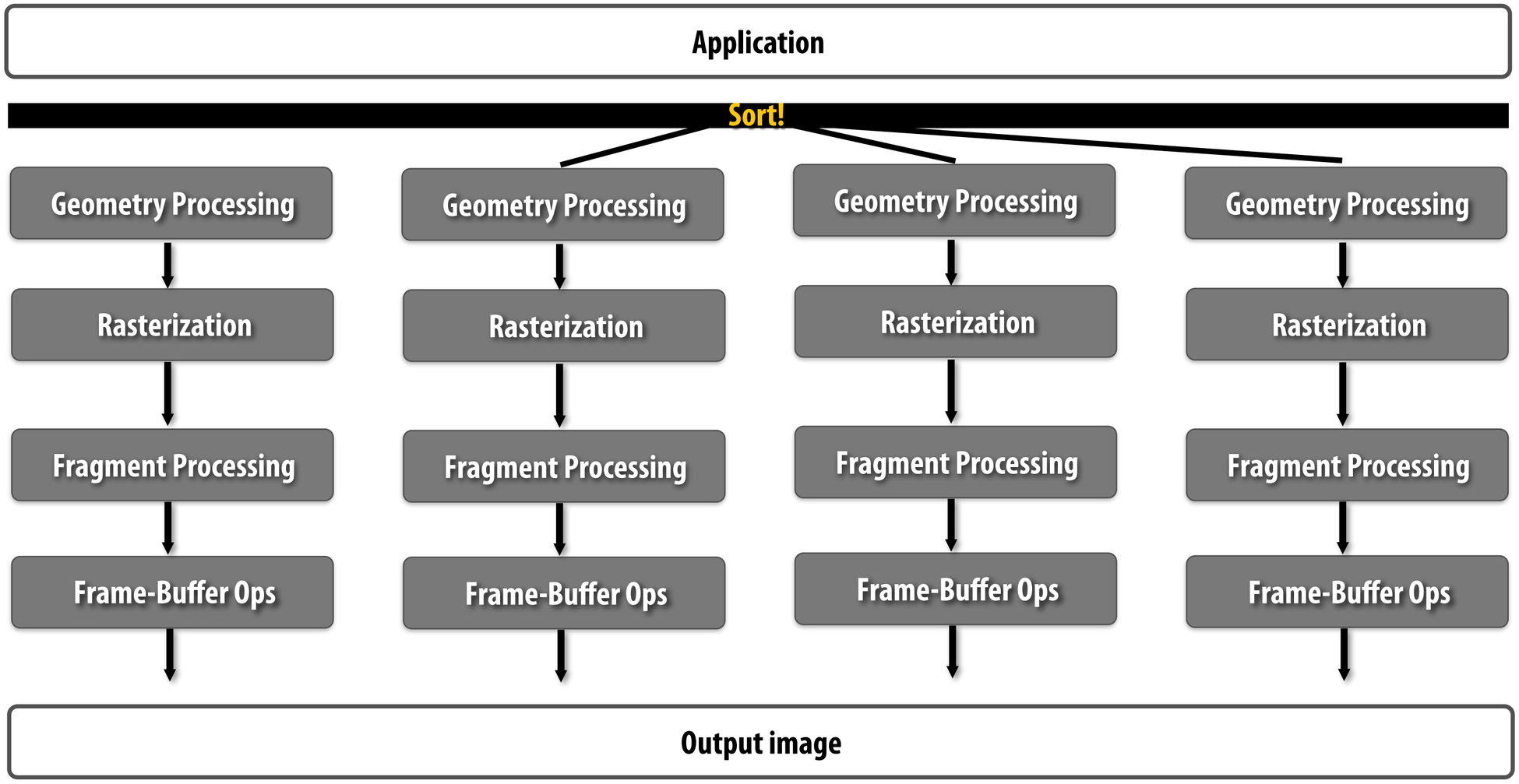


"… the sort is the point where work is sorted based on screen-space location …"
Eldridge's classification defined "sort" as communication between object and image space; "distribute" was object → object; "route" was image → image.

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

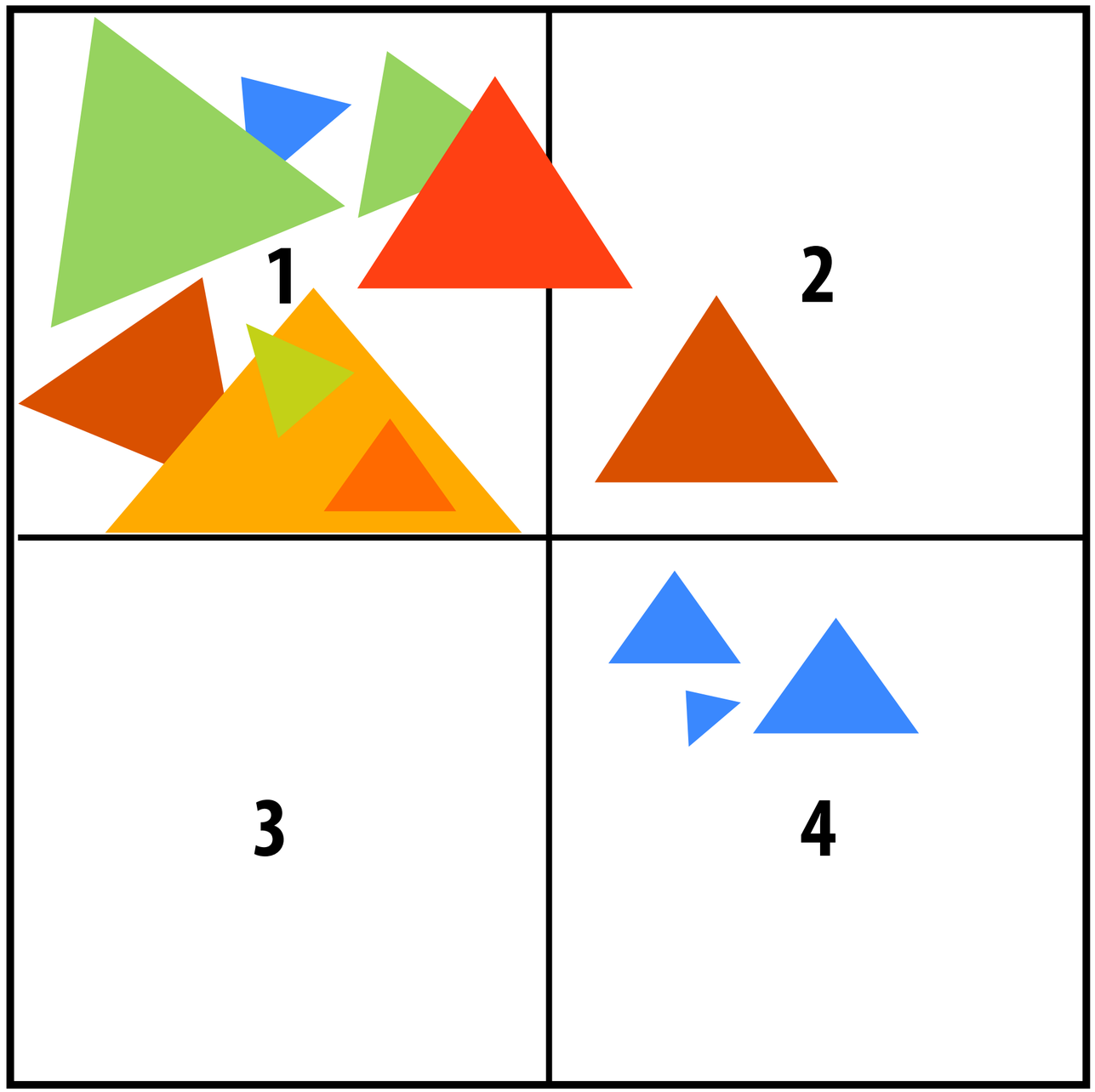
Sort first

Sort first



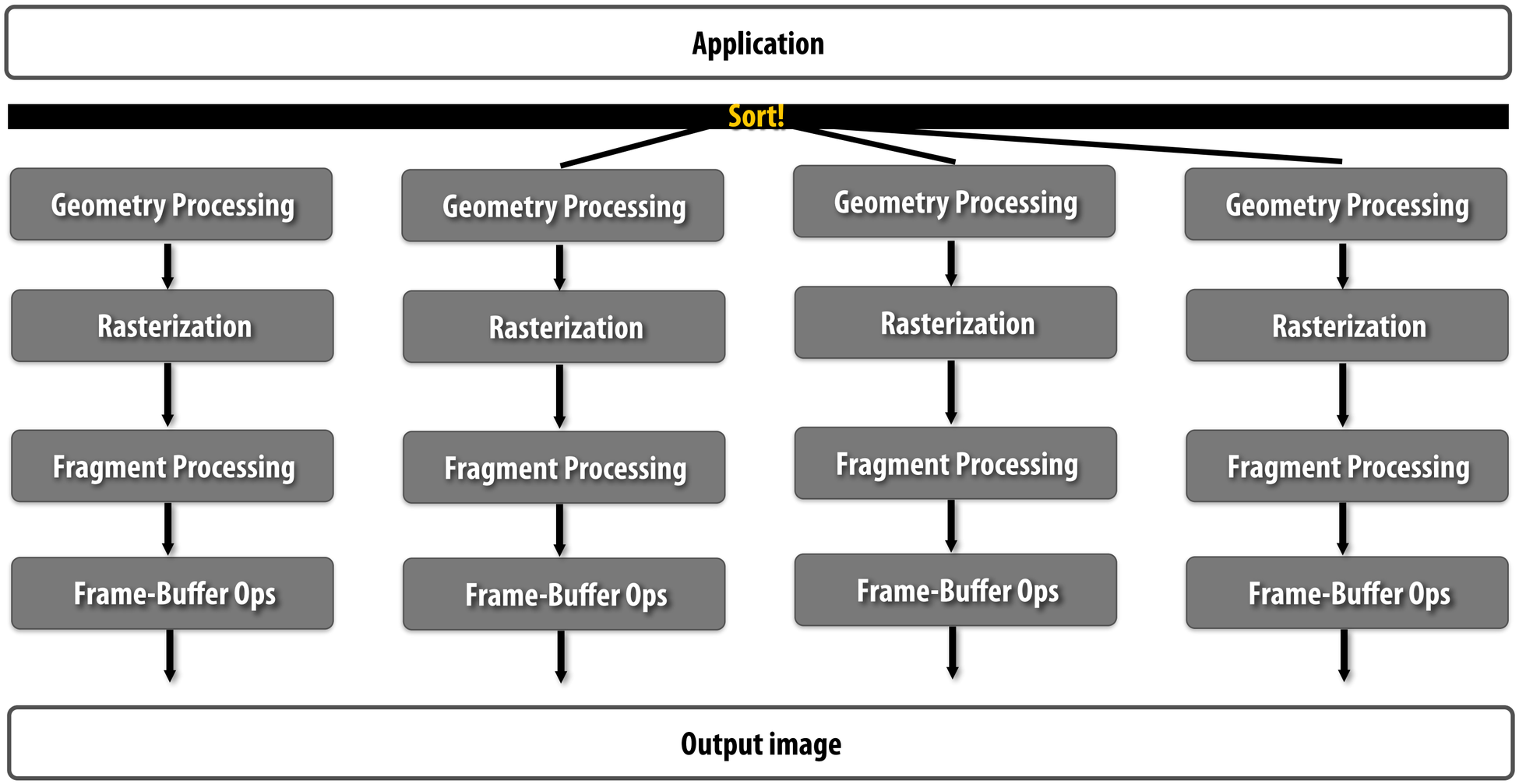
- Assign each replicated pipeline responsibility for a region of the output image
- Do minimal amount of work (compute screen-space vertex positions of triangle) to determine which region(s) each input primitive overlaps

Sort first work partitioning



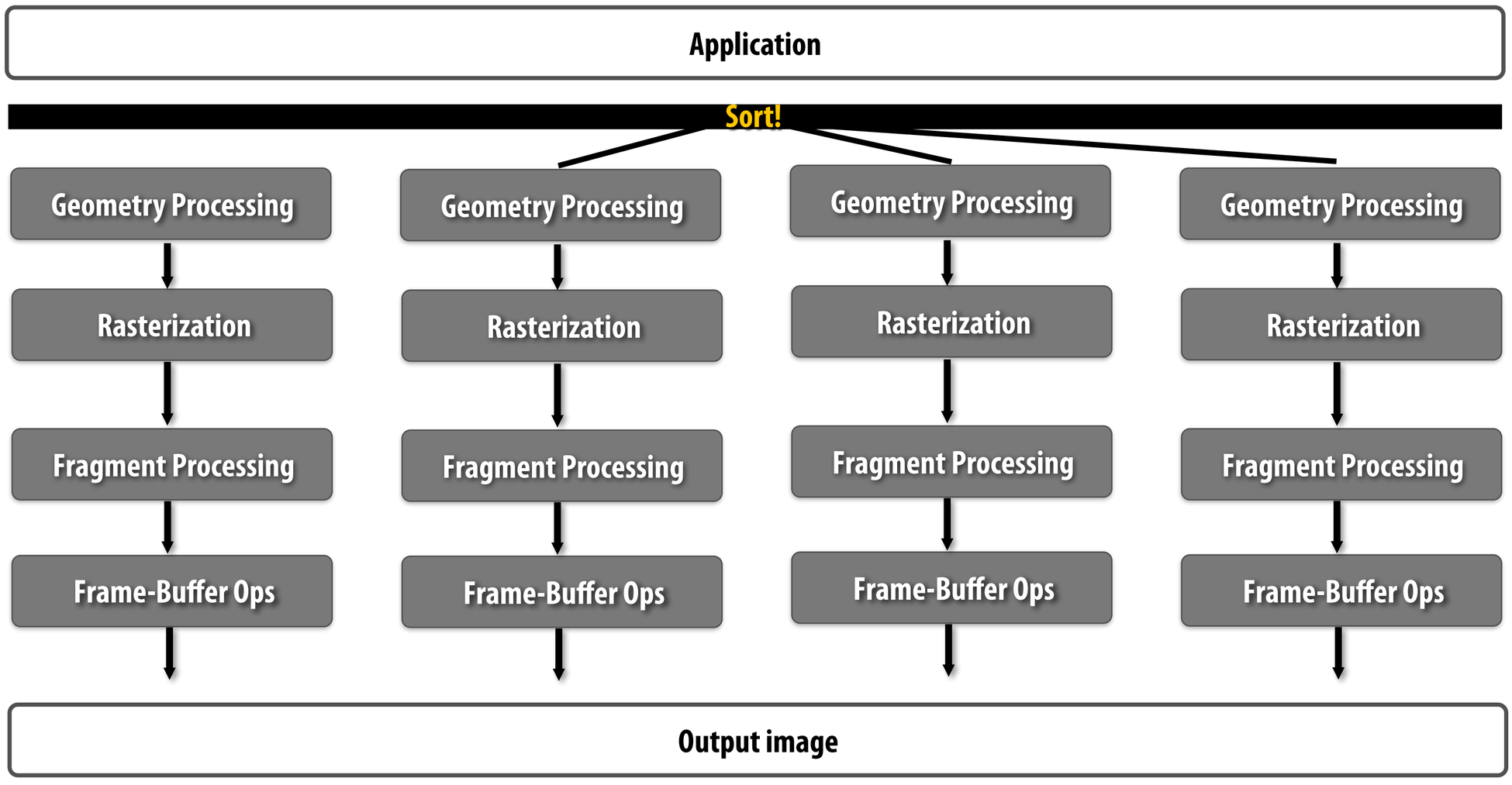
- Partition the primitives to parallel units based on screen overlap

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**

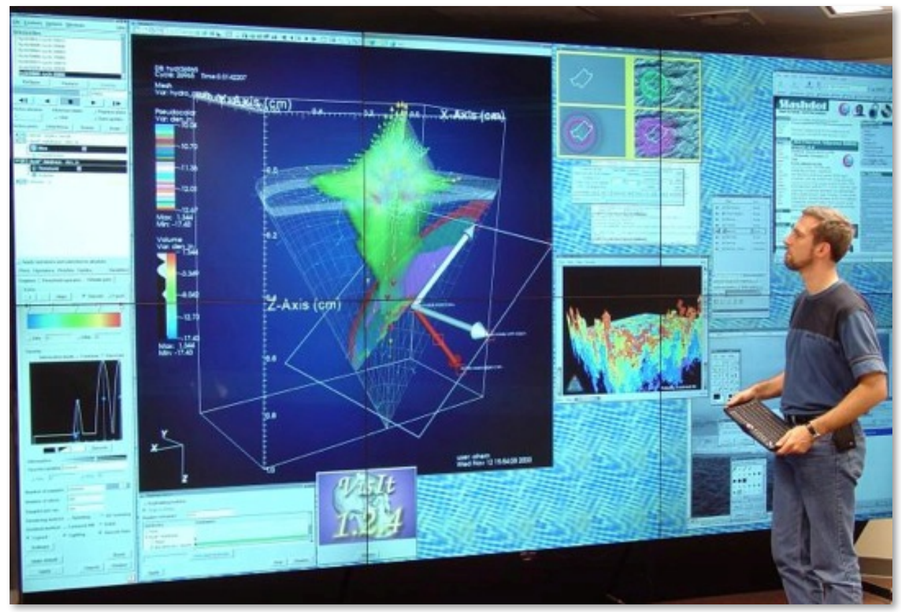
Sort first



- **Bad:**
  - Extra cost of triangle "pre-transformation" (needed to sort)
  - Potential for workload imbalance (one part of screen contains most of scene)
  - "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" sorts primitives to machines
  - Each GPU is a full rendering pipeline
  - (responsible for part of screen)



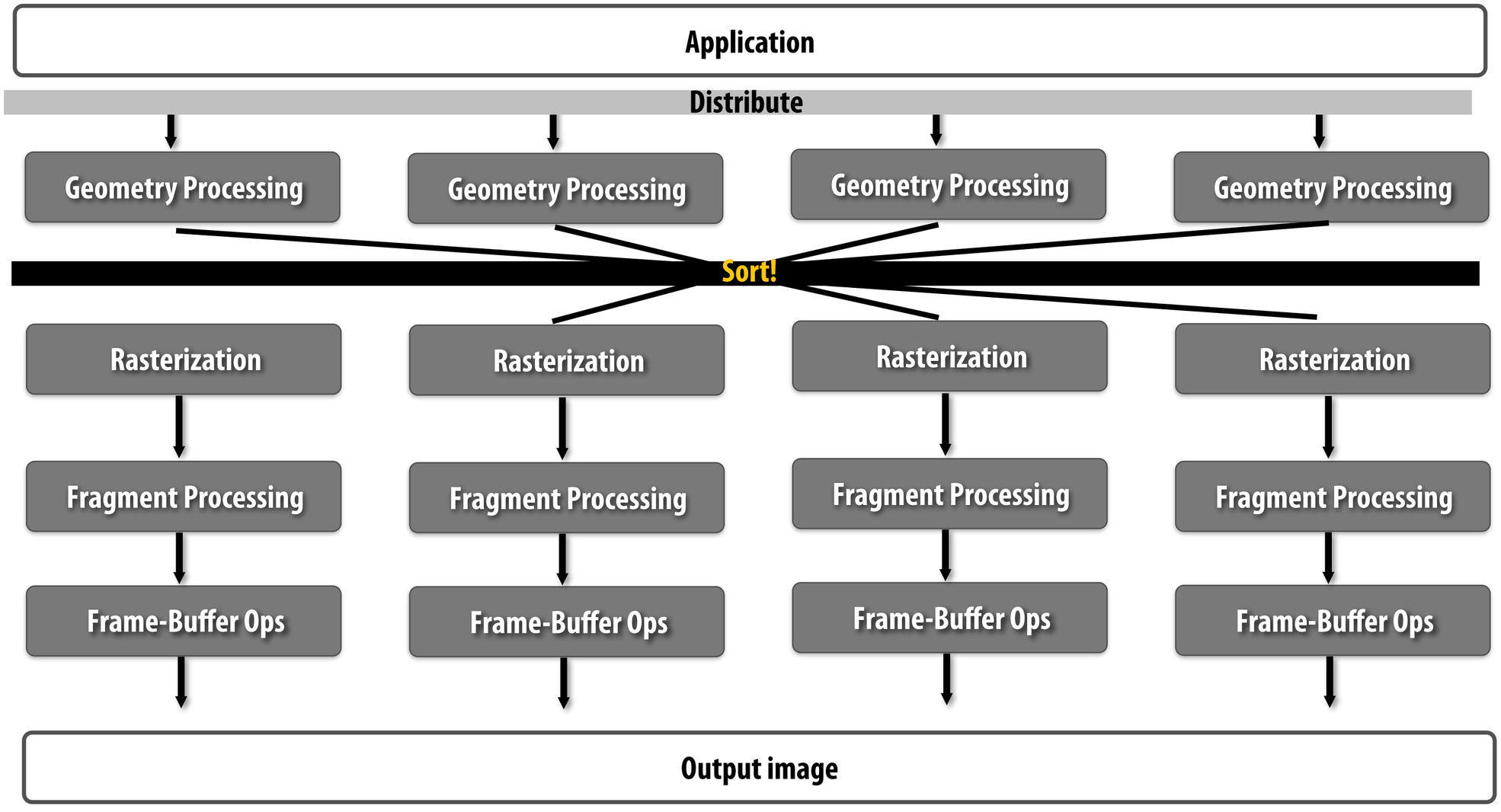
- Pixar's RenderMan
  - Multi-core software renderer
  - Sort surfaces into tiles prior to tessellation

PIXAR's RenderMan

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

Sort middle

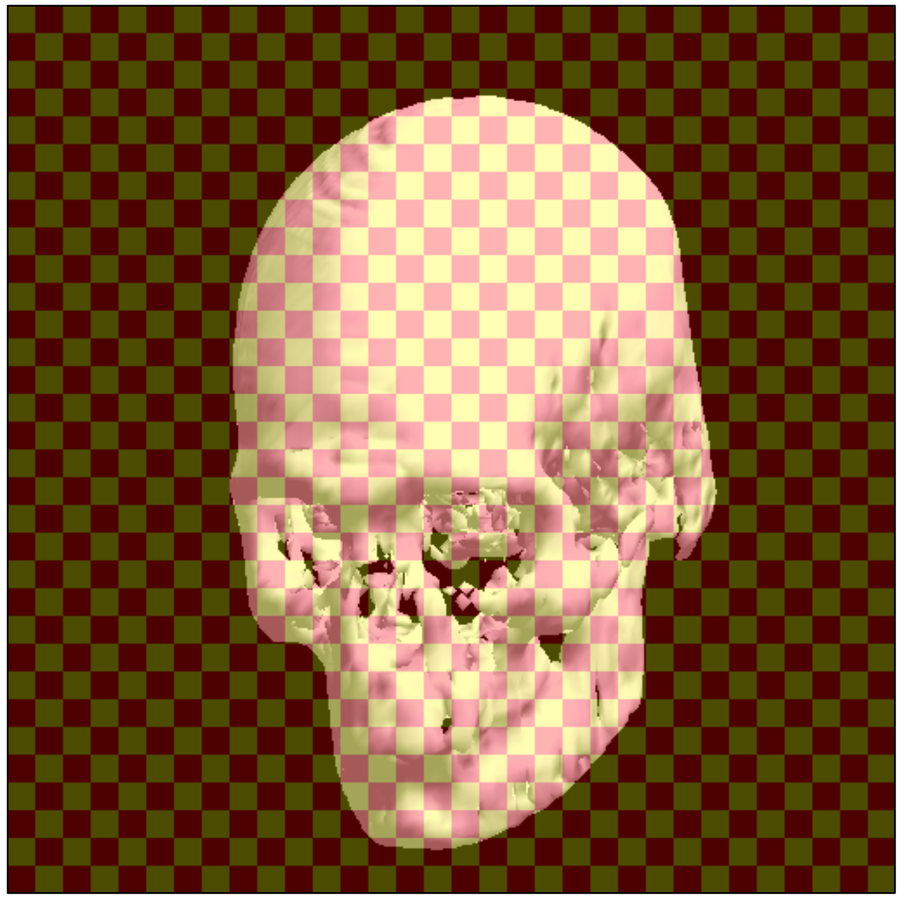
Sort middle



- 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

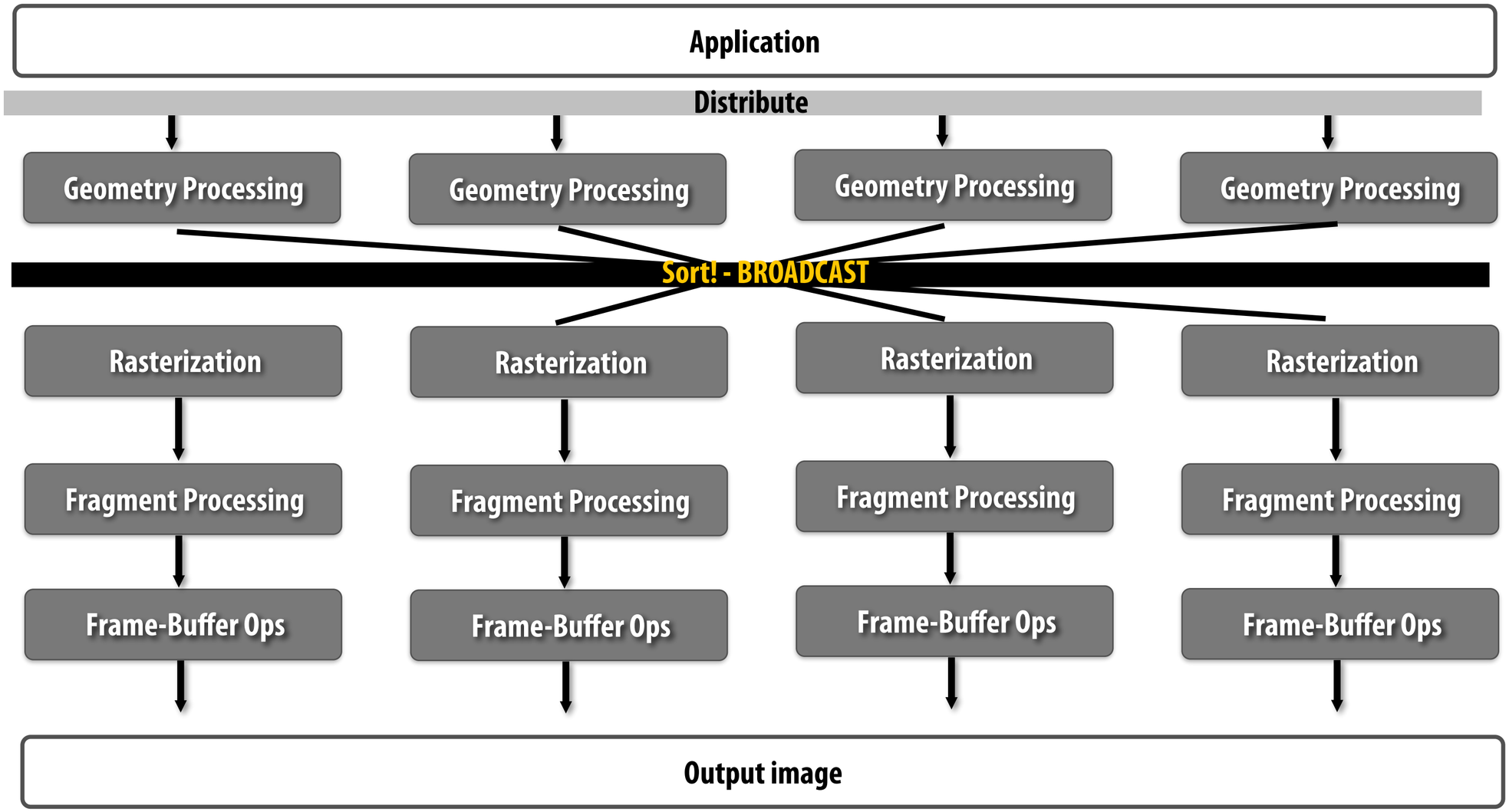
- Decrease chance of one rasterizer processing most of scene
- Most triangles overlap multiple screen regions (often overlap all)



Interleaved mapping

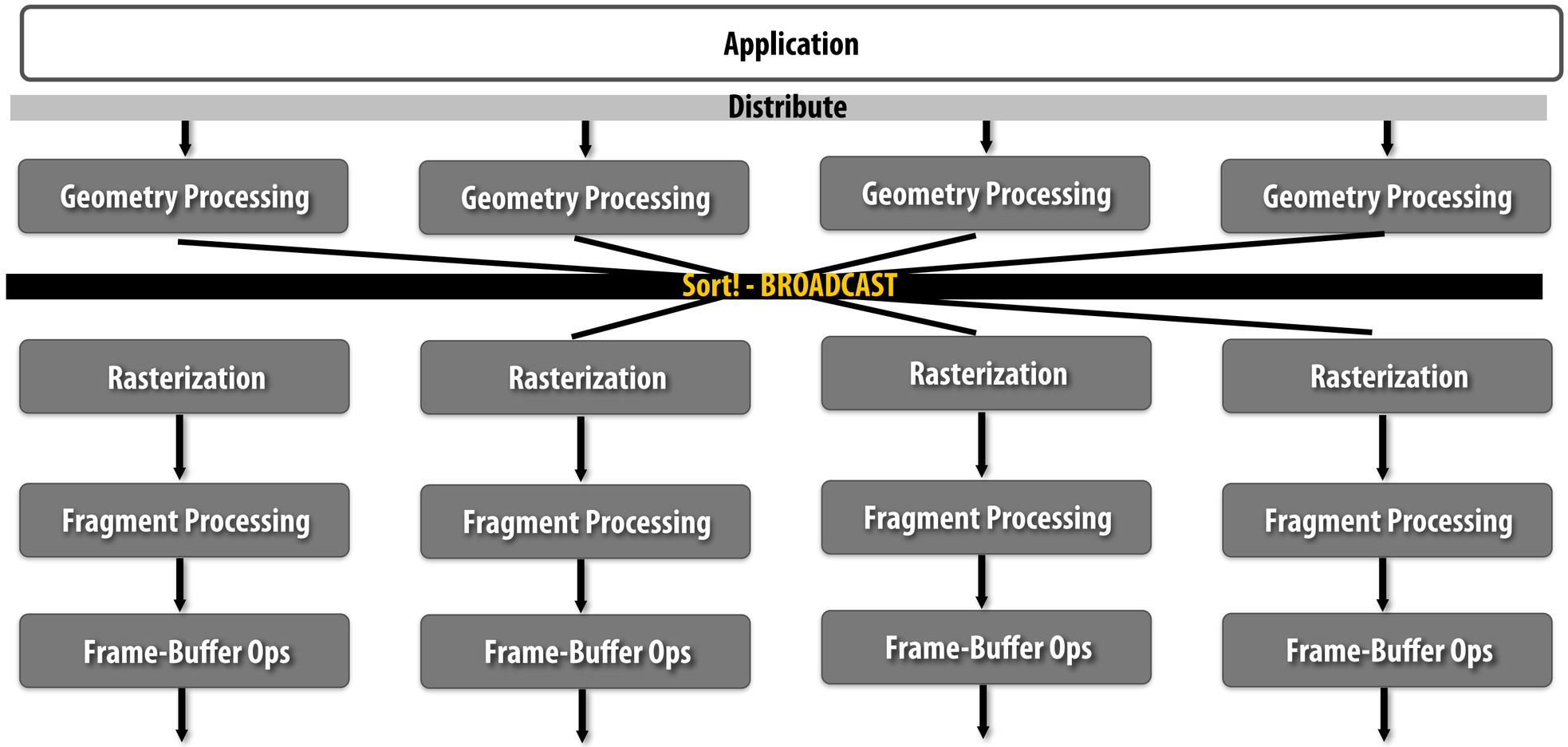
Tiled mapping

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

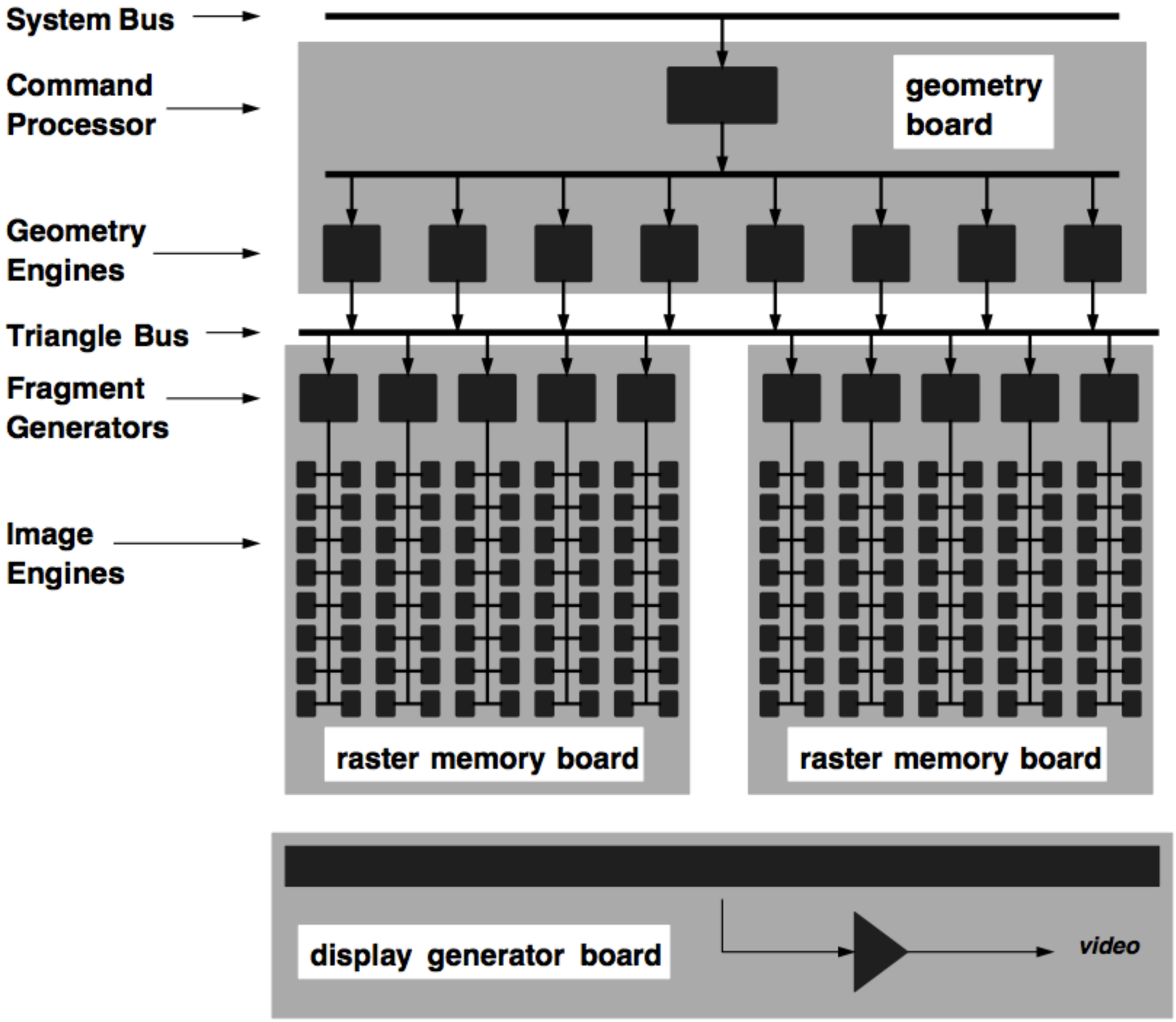
Sort middle interleaved



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

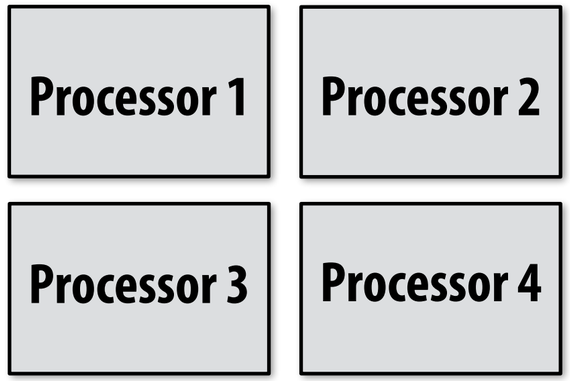
SGI RealityEngine [Akeley 93]

- Sort-middle interleaved design

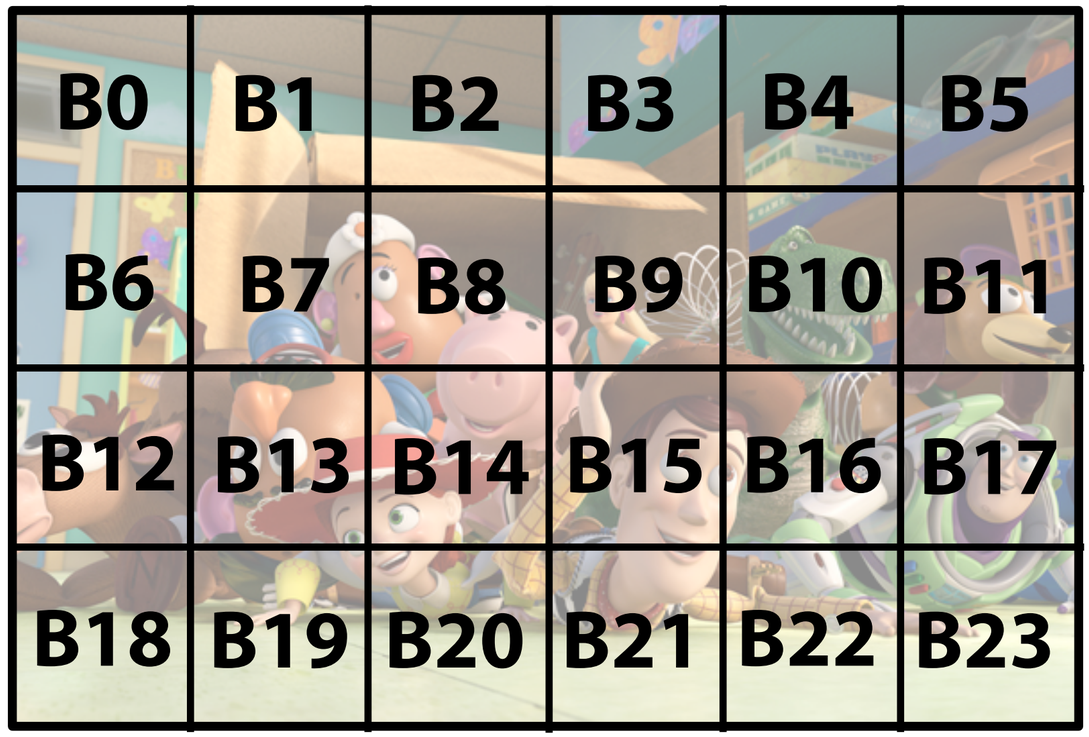


Tiling (a.k.a. “chunking”, “bucketing”)

Two possible strategies:



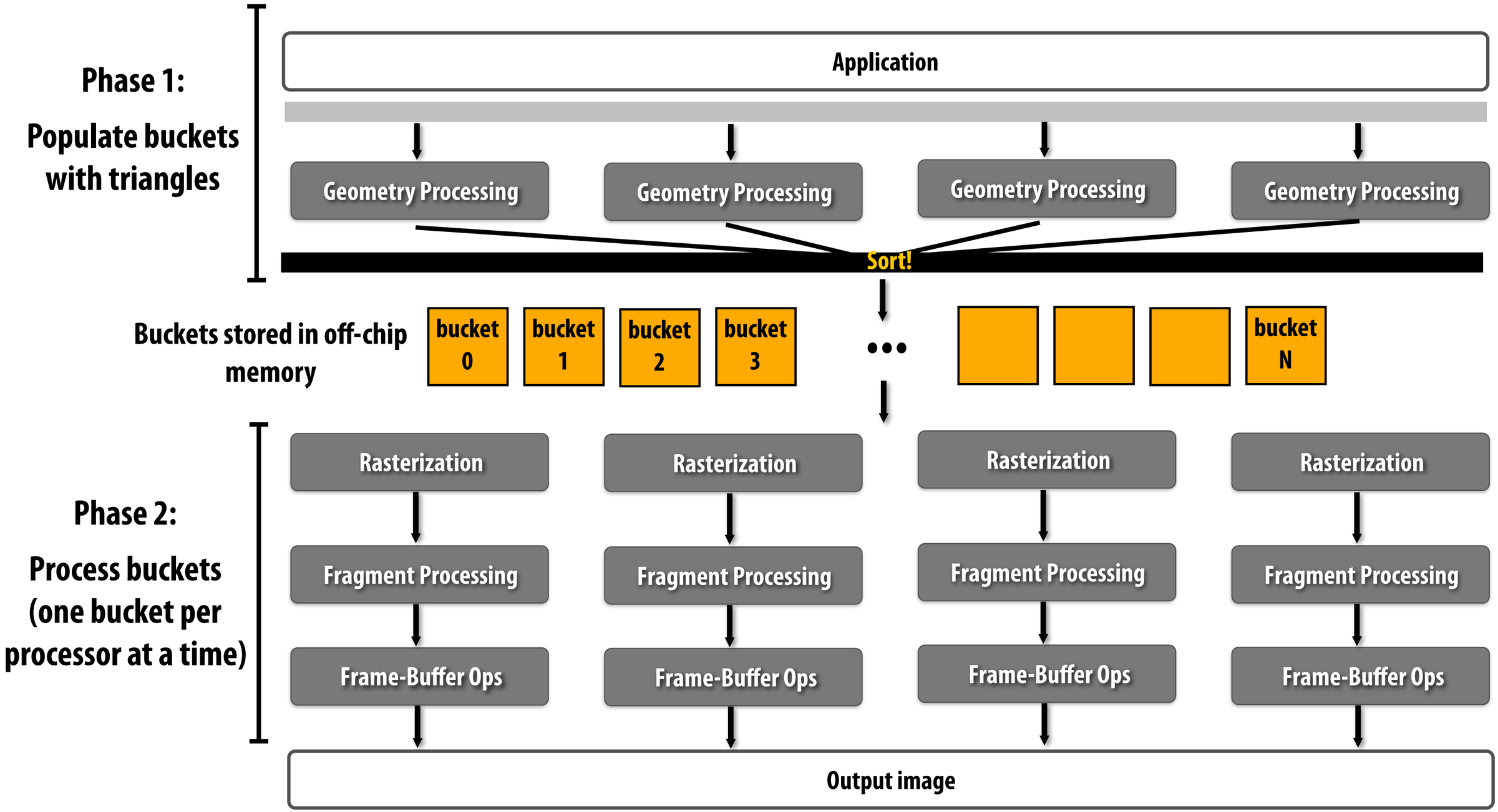
Interleaved (static) assignment to processors



Assignment to buckets

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

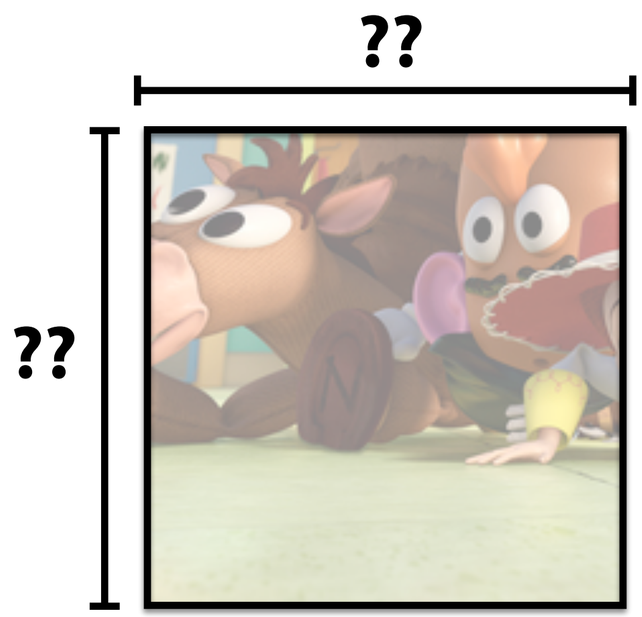
Sort middle tiled (chunked)



- 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

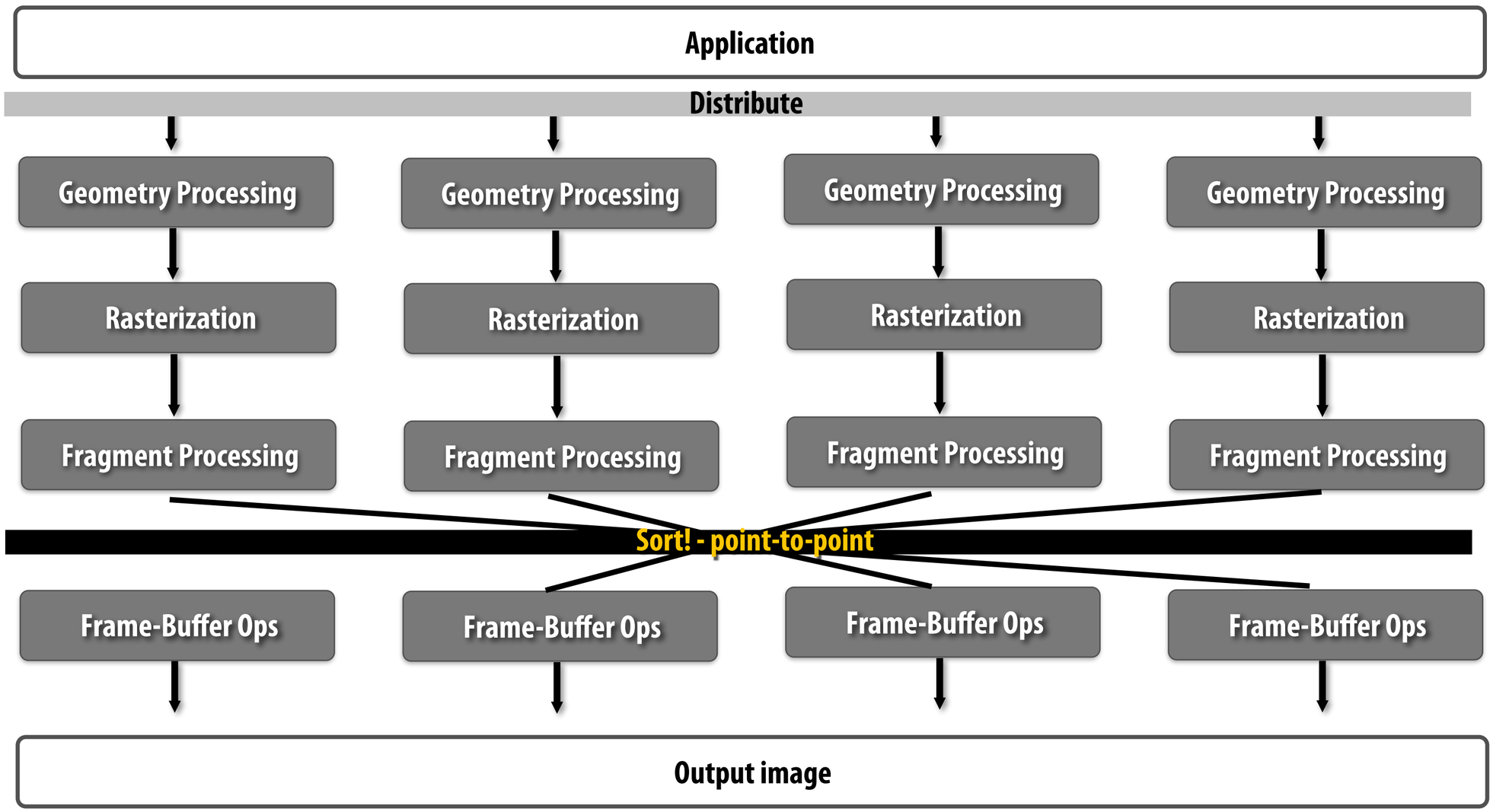
Sort middle tiled (chunked)

- 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



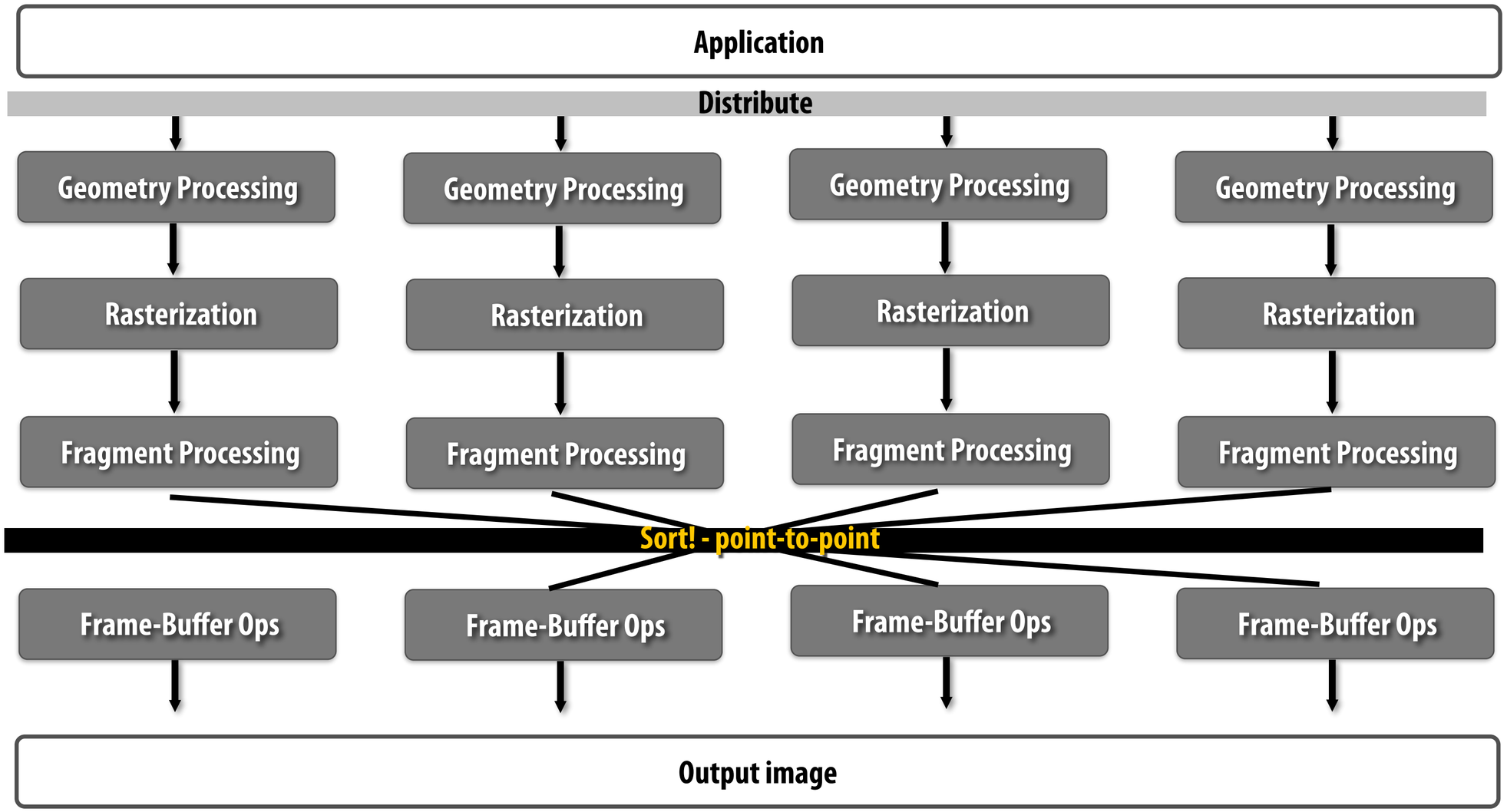
Sort last

Sort last fragment



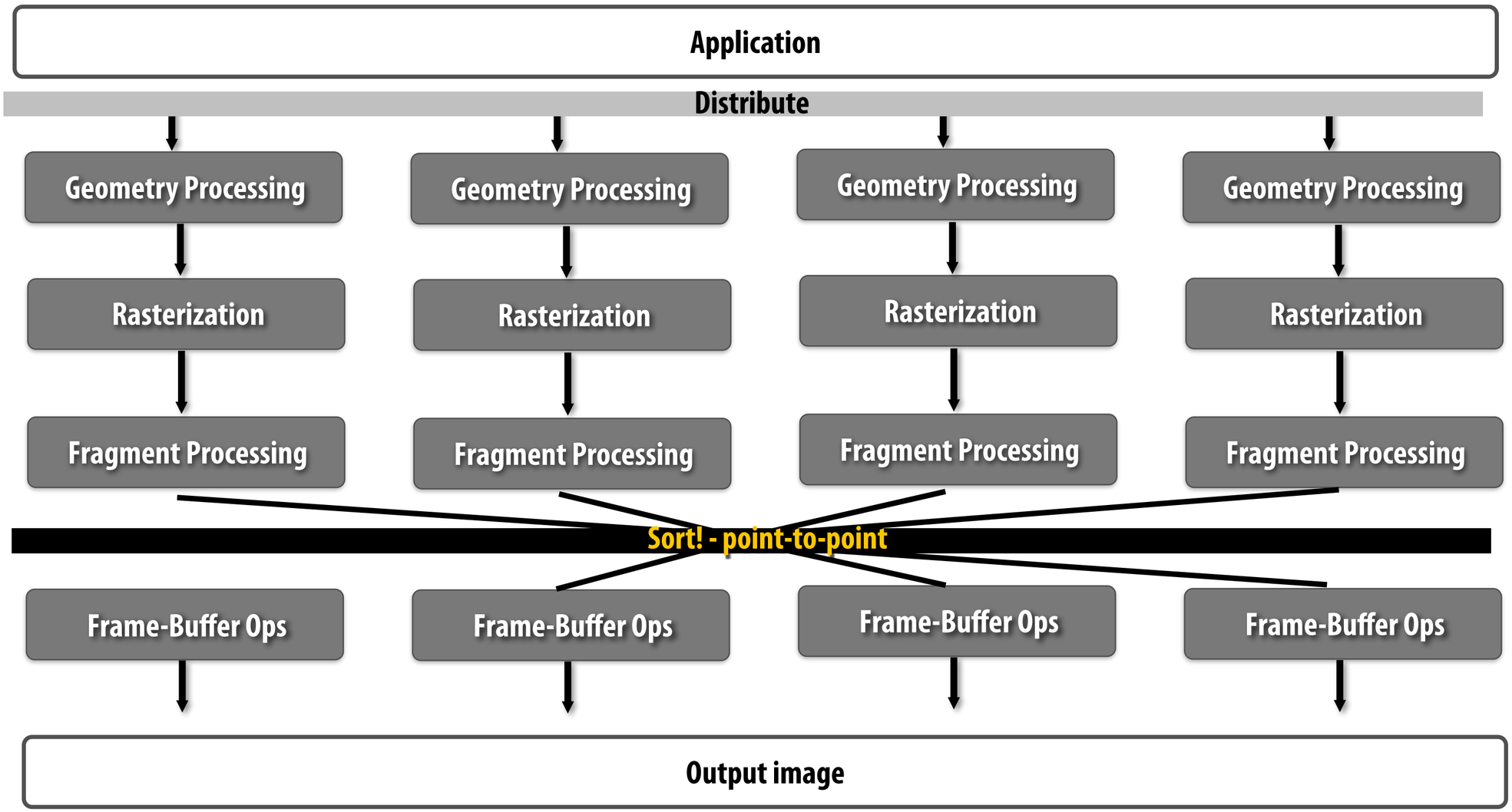
- Distribute primitives to top of pipelines (e.g., round robin)
- Sort after fragment processing based on (x,y) position of fragment

Sort last fragment



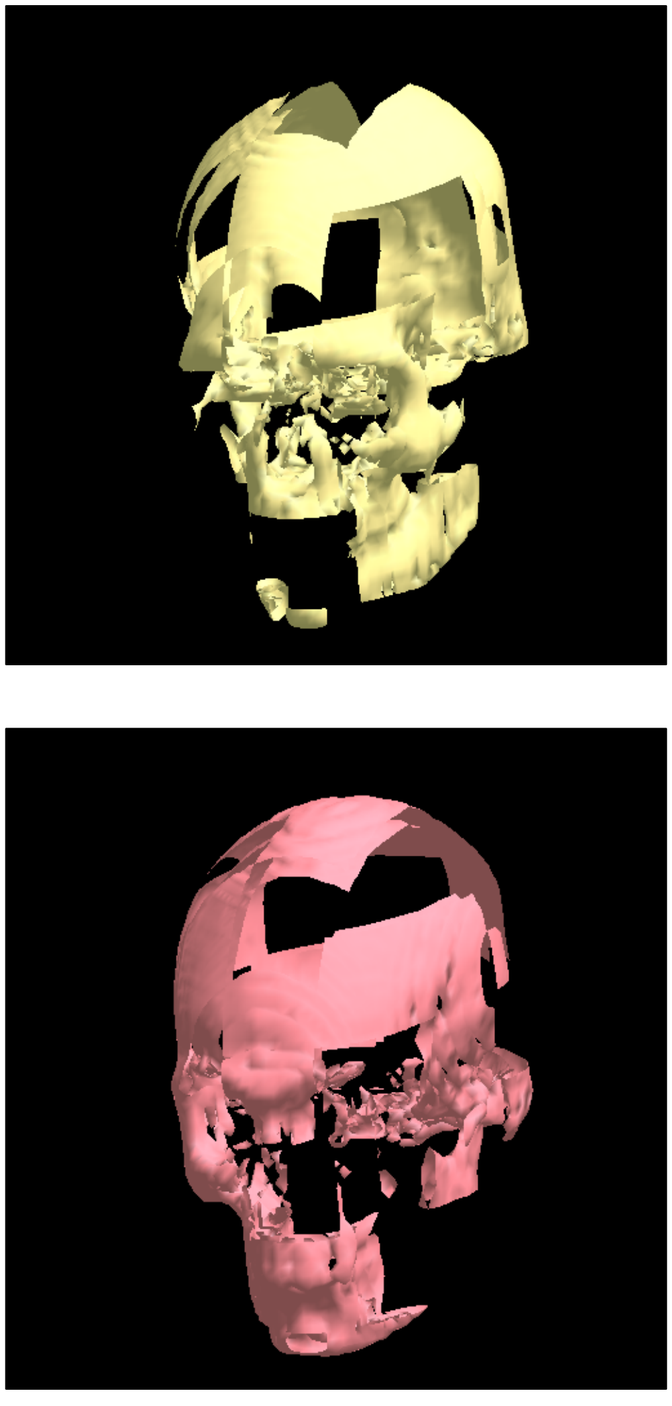
- **Good:**
  - No redundant geometry processing or rasterization (but early z-cull is a problem)
  - Point-to-point communication during sort
  - Interleaved pixel mapping results in good workload balance for frame-buffer ops

Sort last fragment

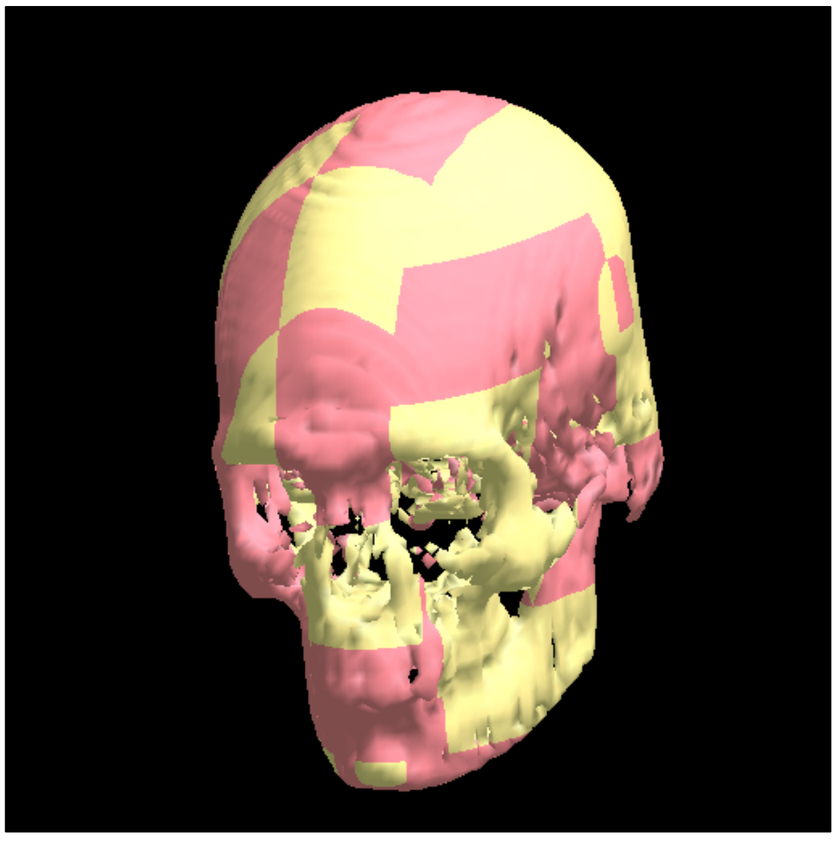


- **Bad:**
  - Pipelines may stall due to primitives of varying size (due to order requirement)
  - Bandwidth scaling: many more fragments than triangles
  - Hard to implement early occlusion cull (more bandwidth challenges)

Sort last image composition (here, Z-Composition)

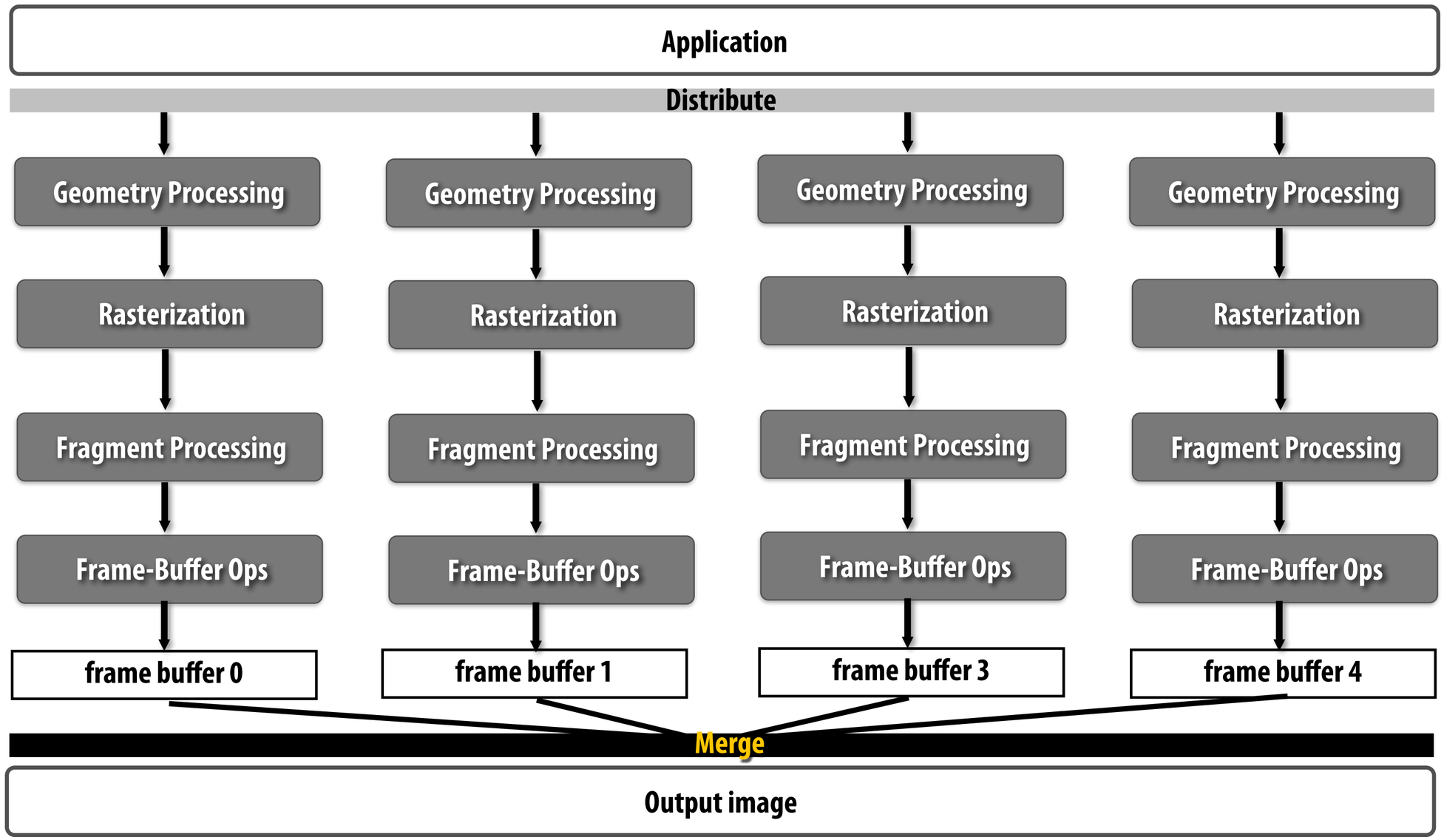


Z comp



Other combiners possible

Sort last image composition

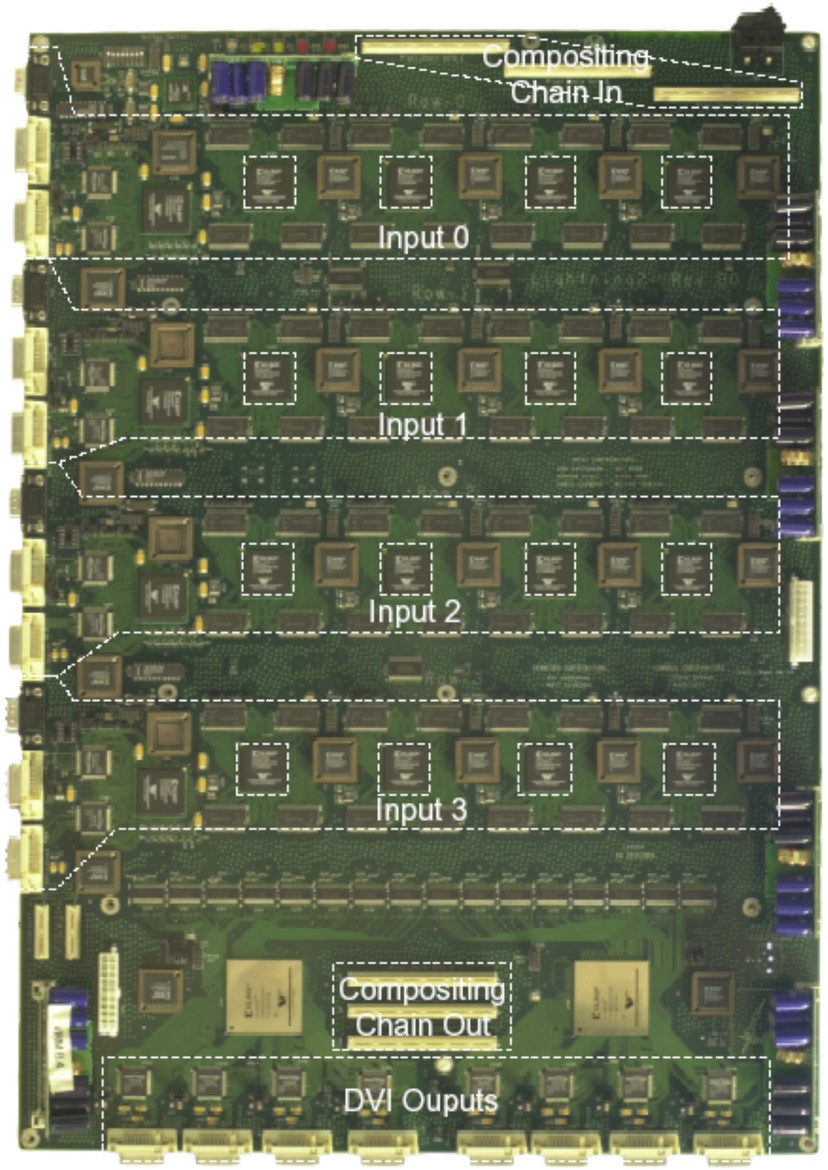


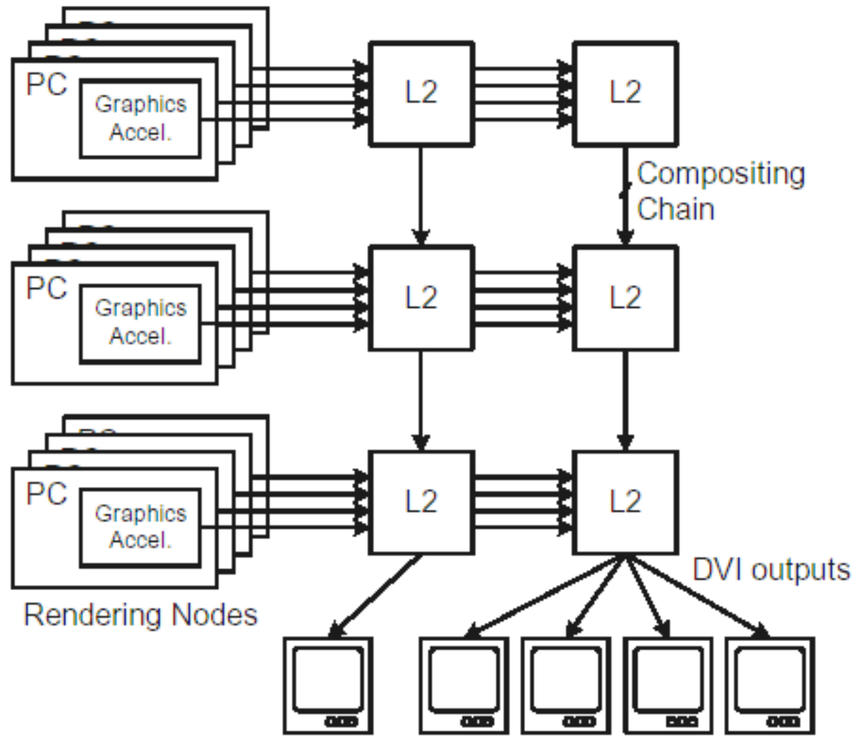
- Each pipeline renders some fraction of the geometry in the scene
- Combine the color buffers, according to depth into the final image

Sort last image composition

- Breaks 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
  - Communication proportional to screen size not geometry

Lightning-2 (Image Composition HW)

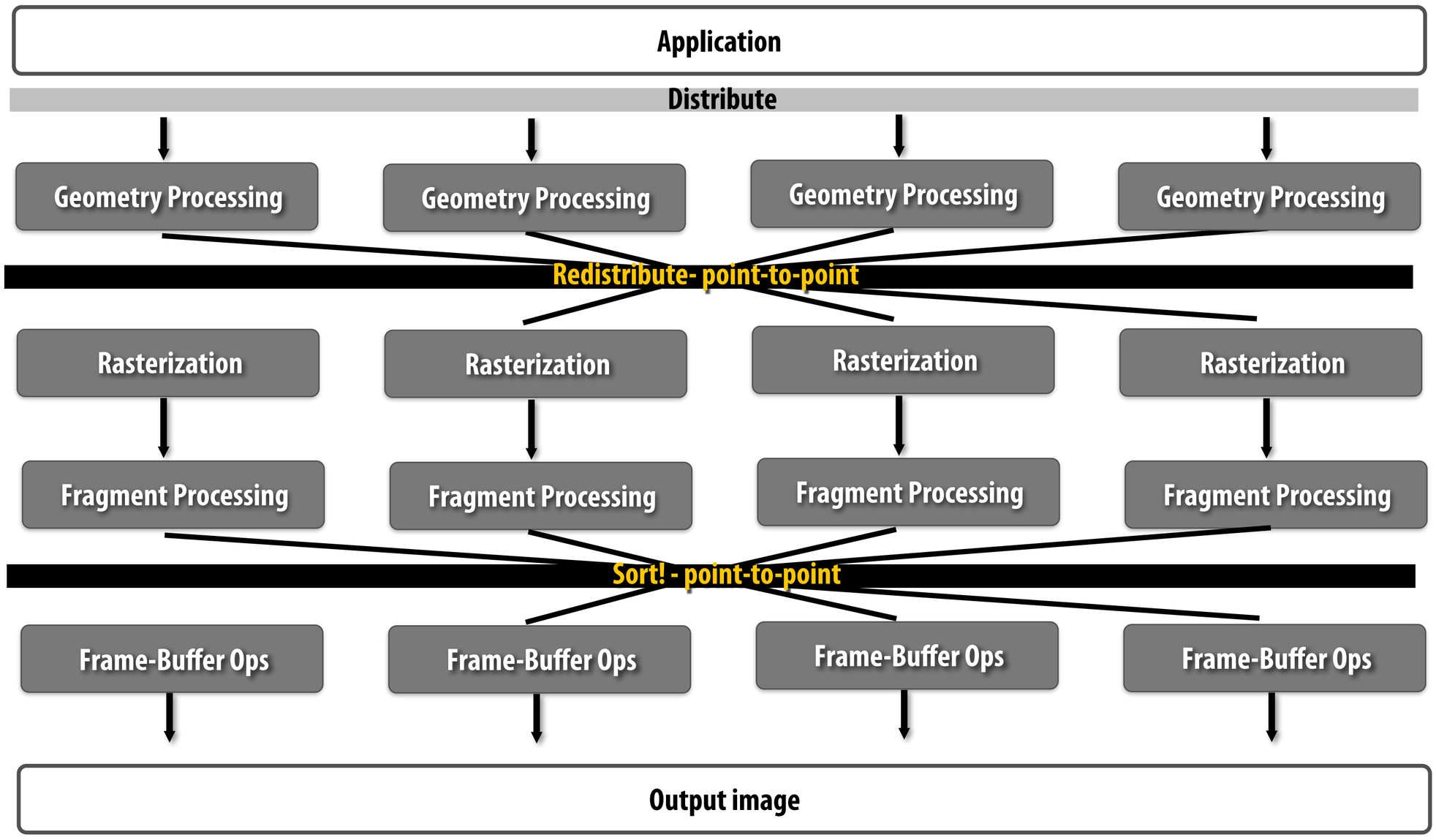




Sort everywhere (How modern high-end GPUs are scheduled)

[Eldridge 00]

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

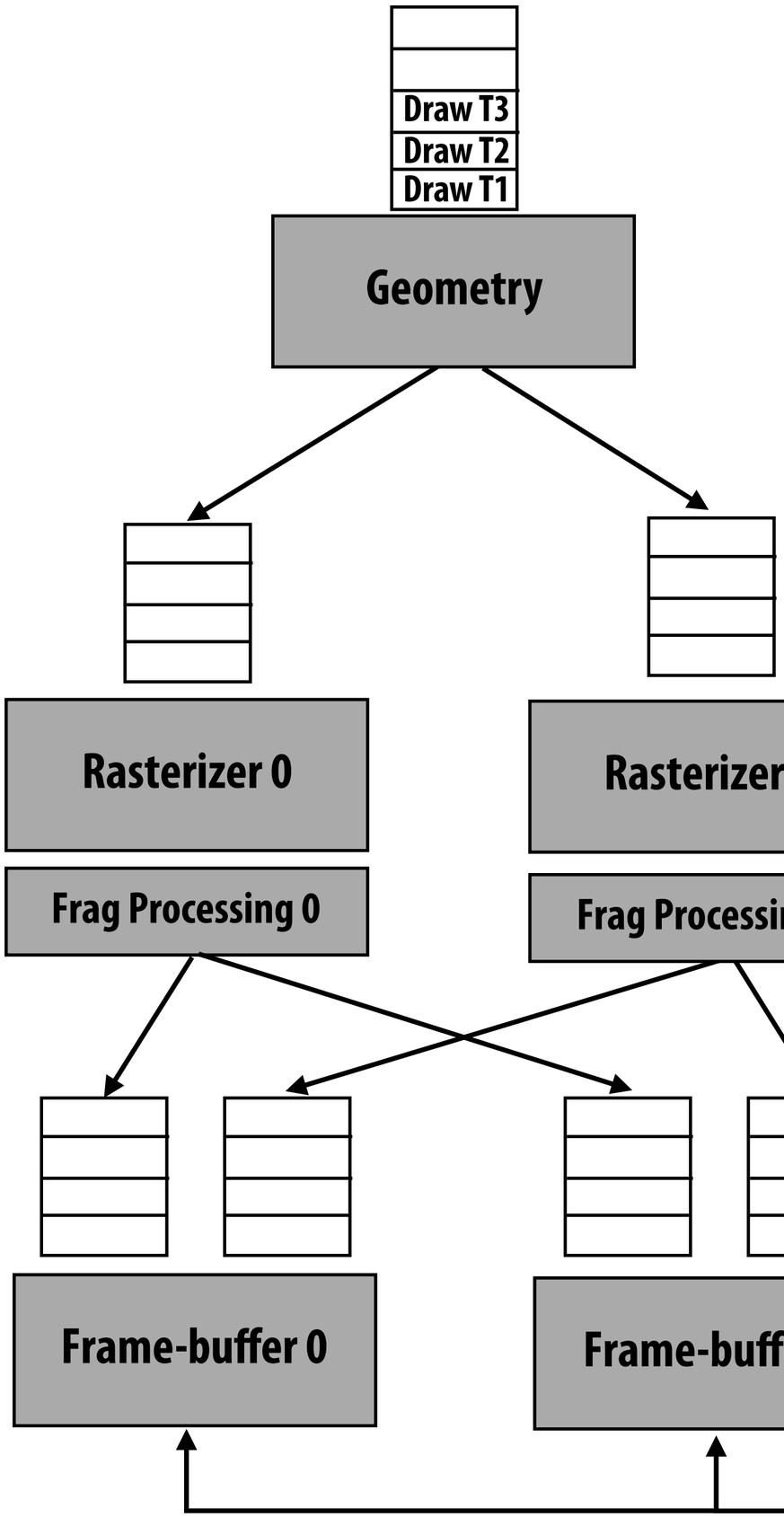
Outline

- Terminology and Strategies
- Communication Structures / Costs
- Taxonomy
- Scheduling

Implementing sort everywhere

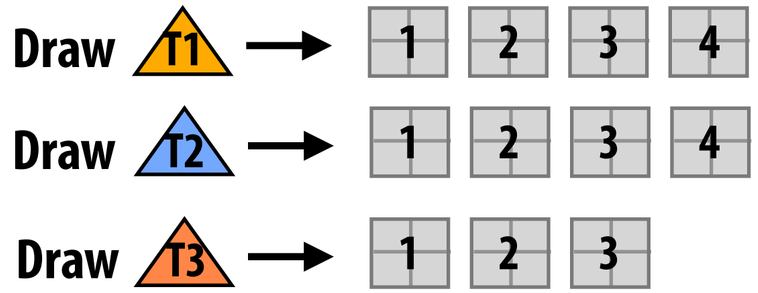
(Challenge: rebalancing work at multiple places in the graphics pipeline to achieve efficient parallel execution, while maintaining triangle draw order)

Starting state: draw commands enqueued for pipeline

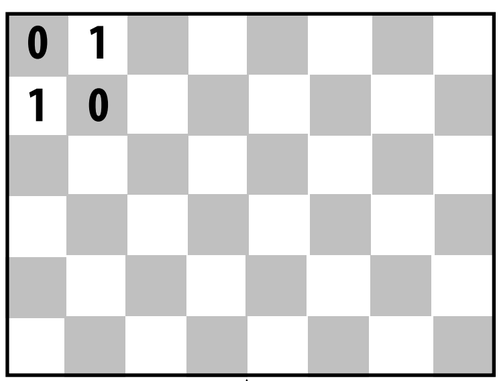


Input: three triangles to draw

(fragments to be generated for each triangle by rasterization are shown below)

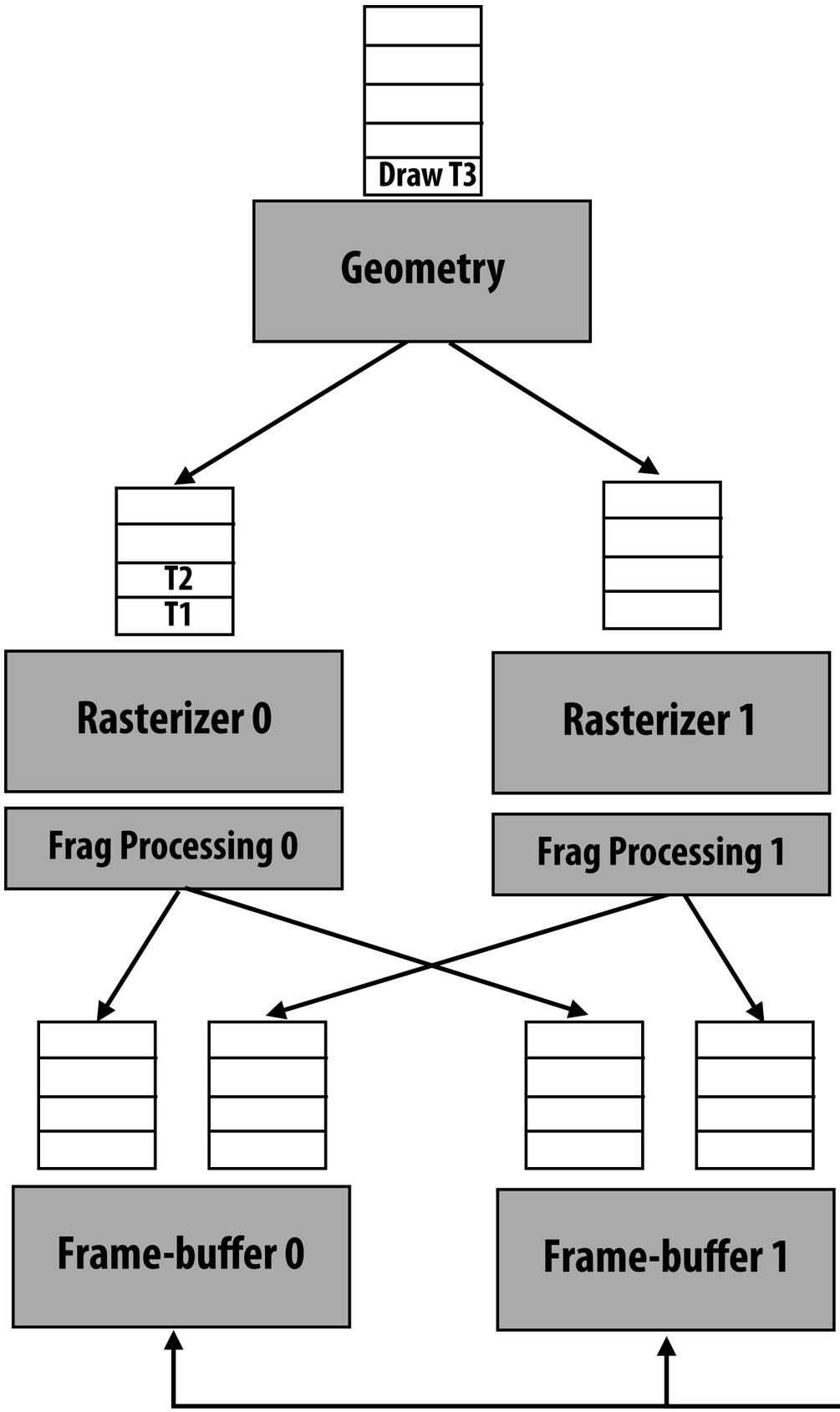


Assume batch size is 2 for assignment to rasterizers.

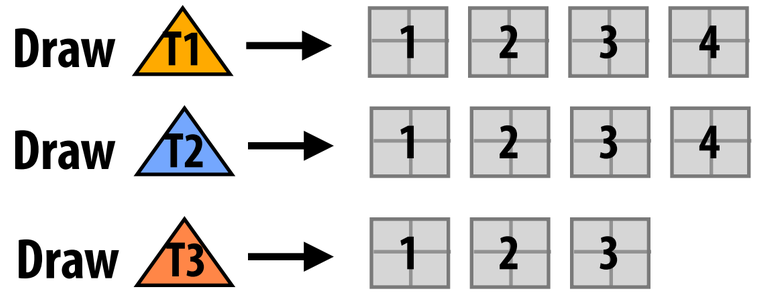


Interleaved render target

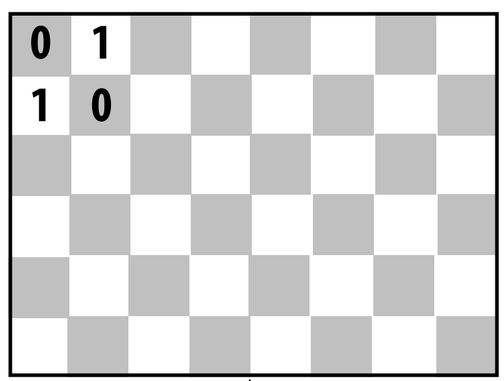
After geometry processing, first two processed triangles assigned to rast 0



Input:

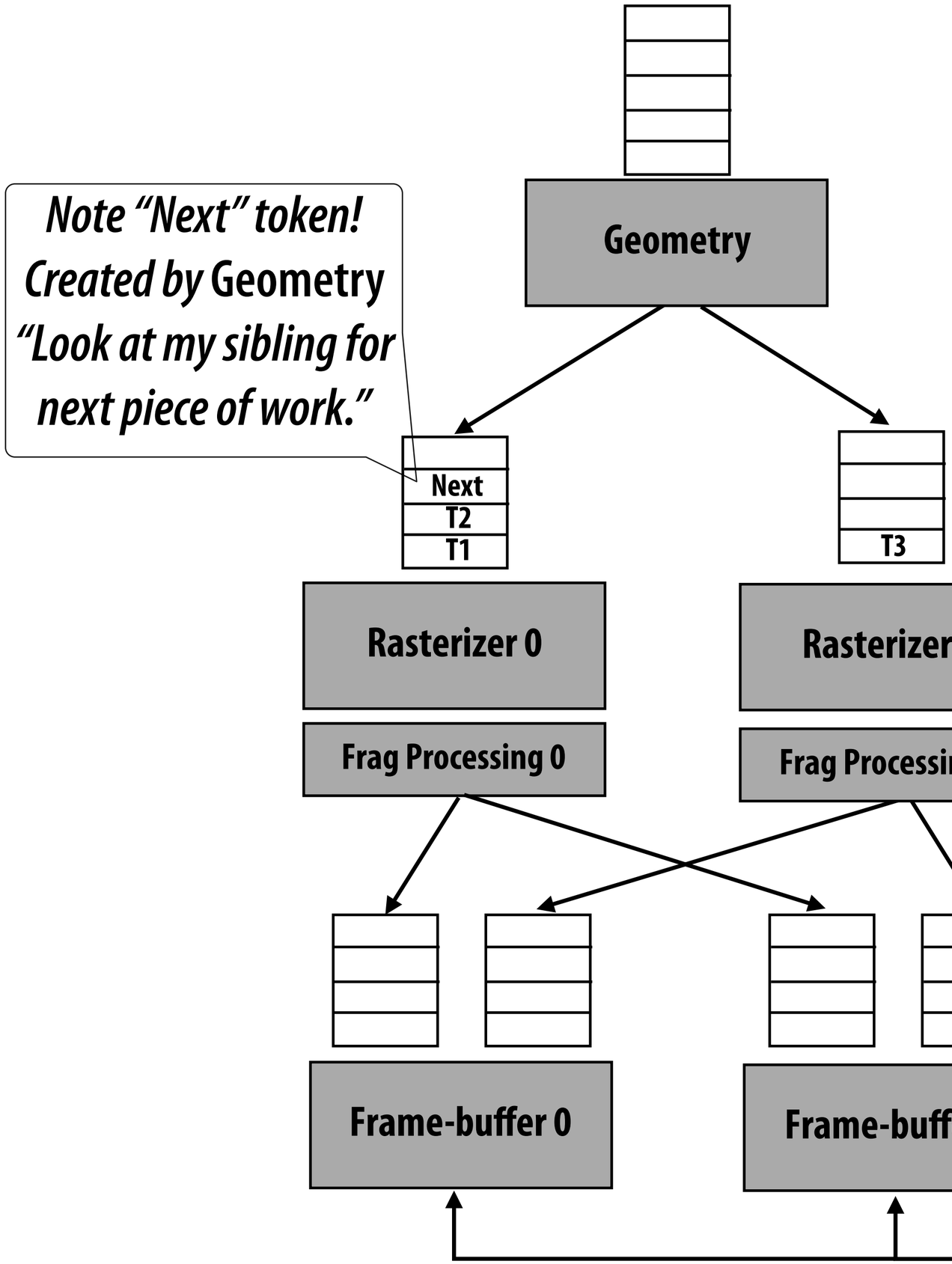


Assume batch size is 2 for assignment to rasterizers.

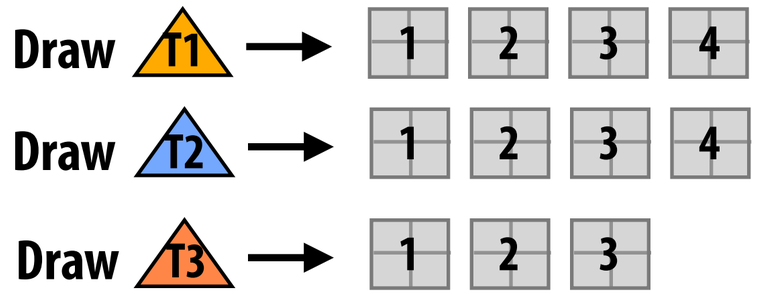


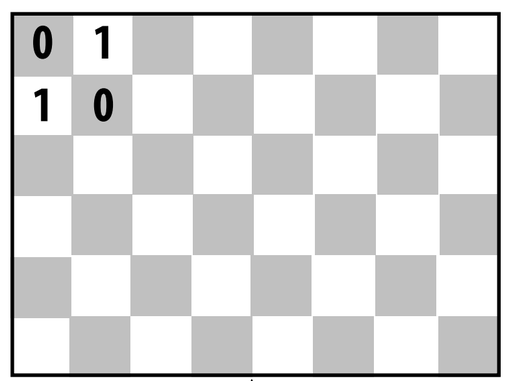
Interleaved render target

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



Input:

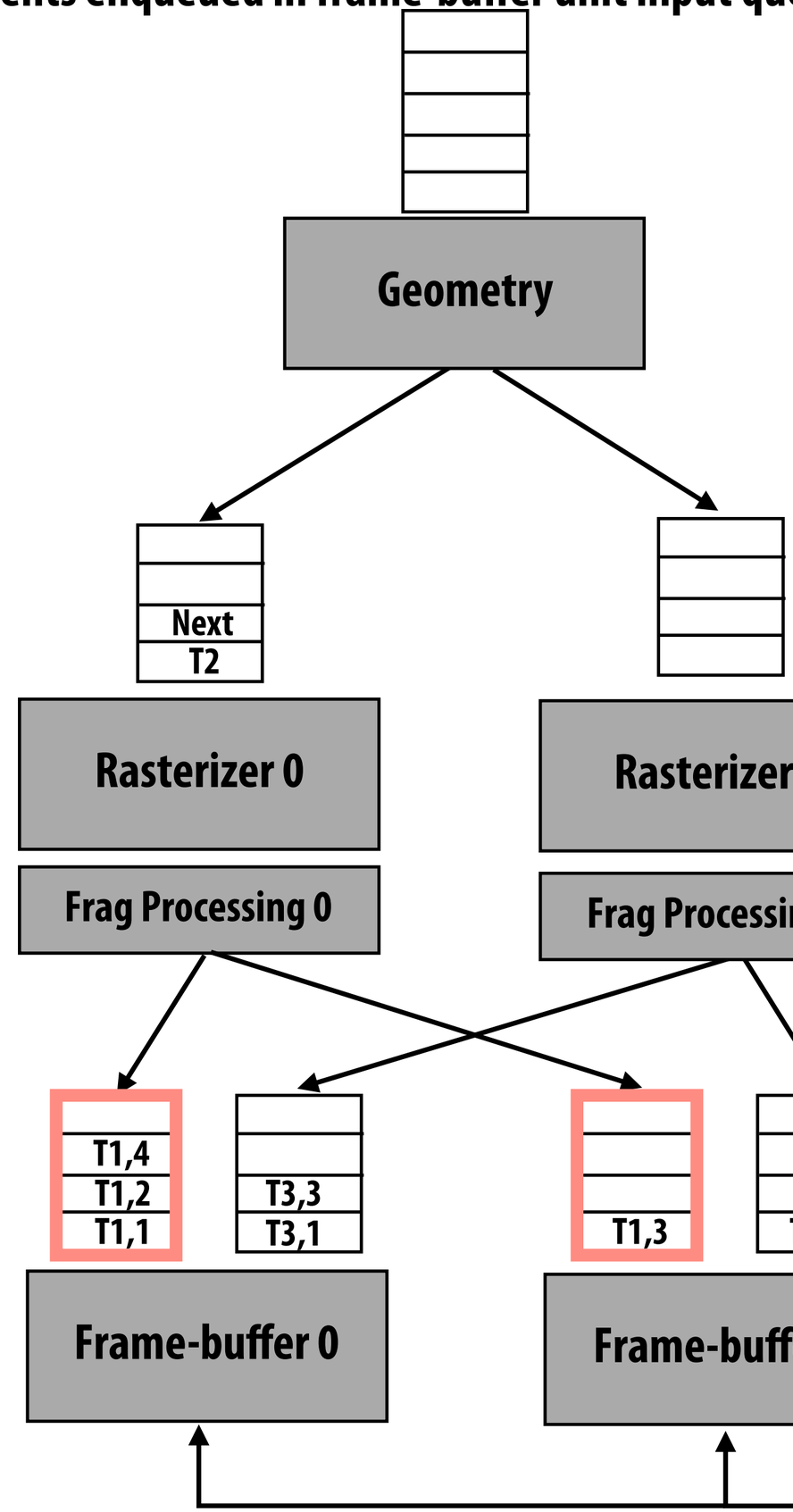




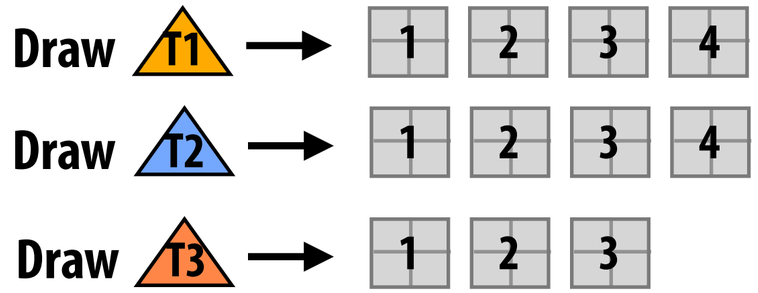
Interleaved render target

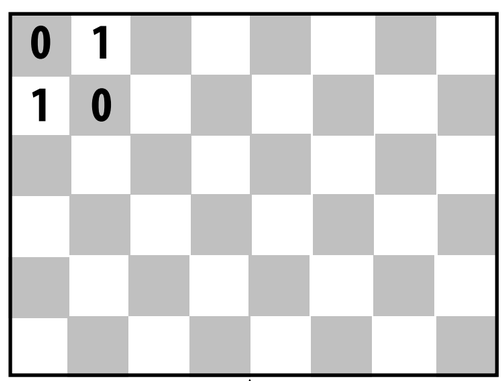
Rast 0 and rast 1 can process T1 and T3 simultaneously

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



Input:

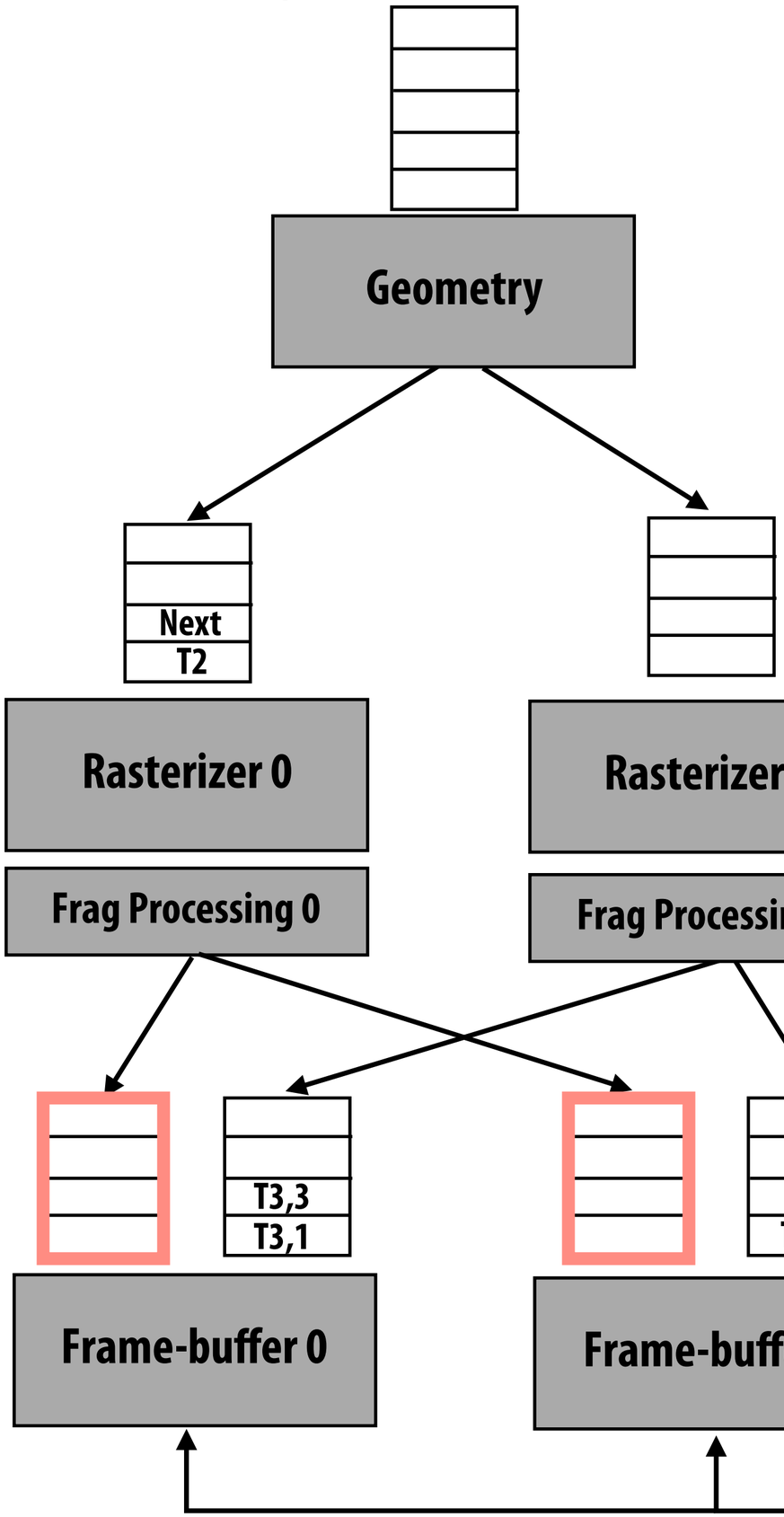




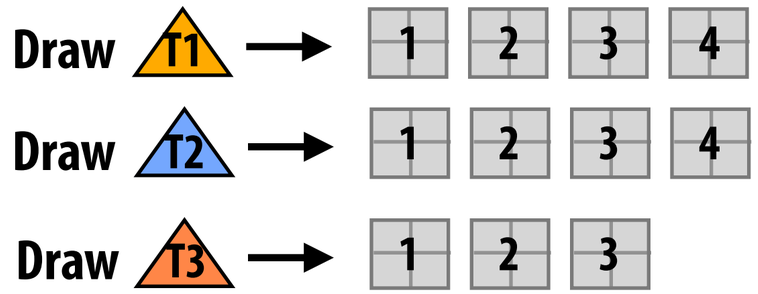
Interleaved render target

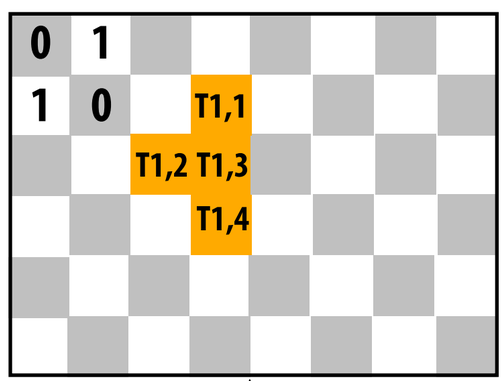
FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)



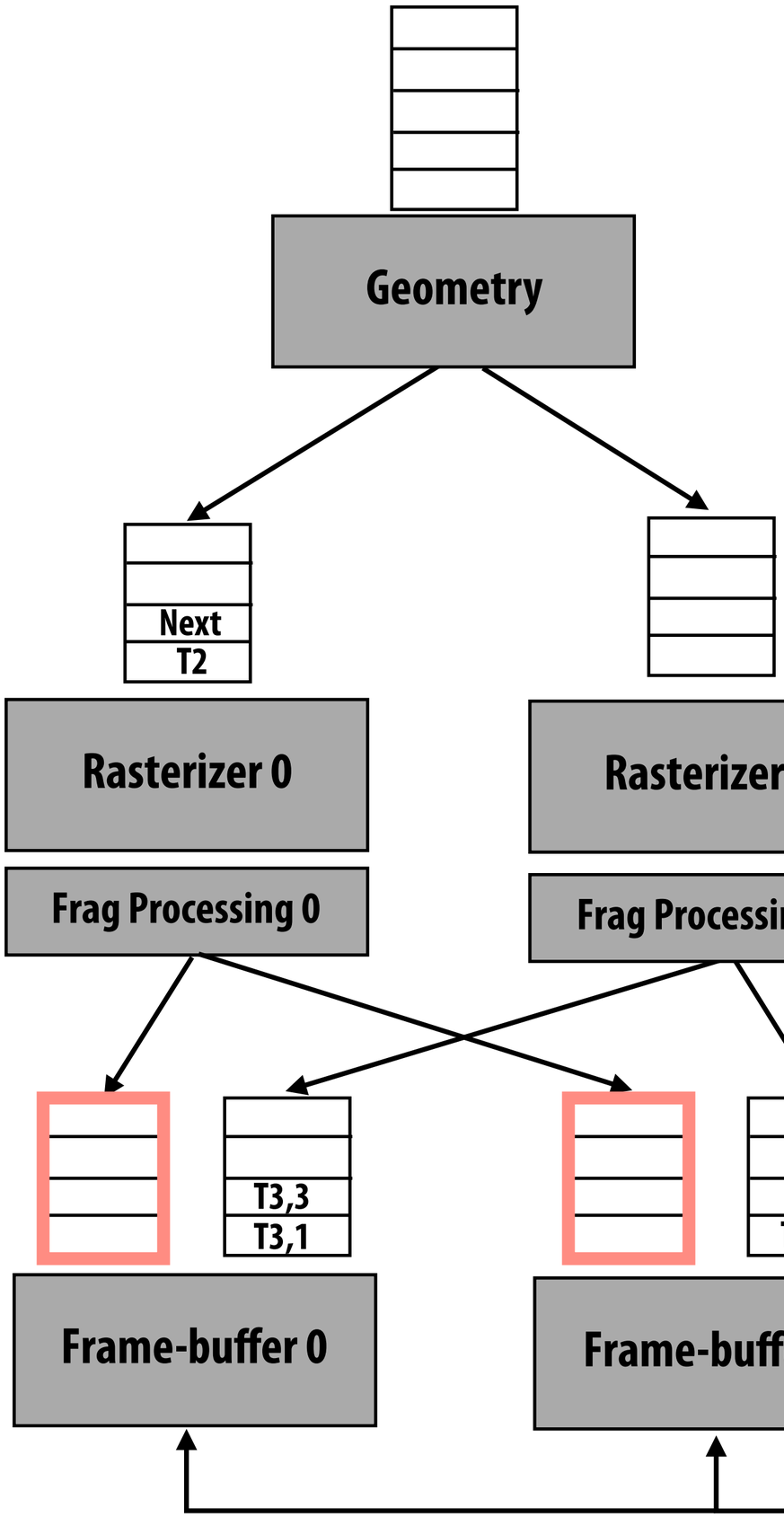
Input:



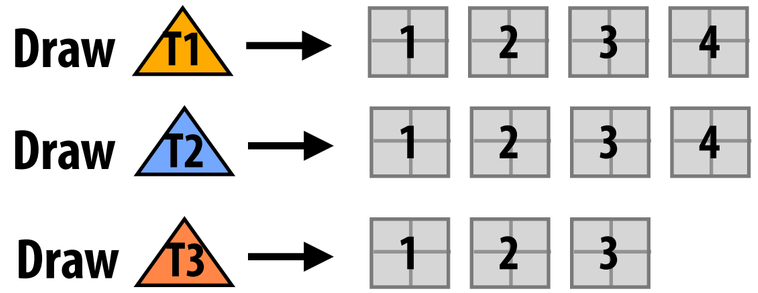


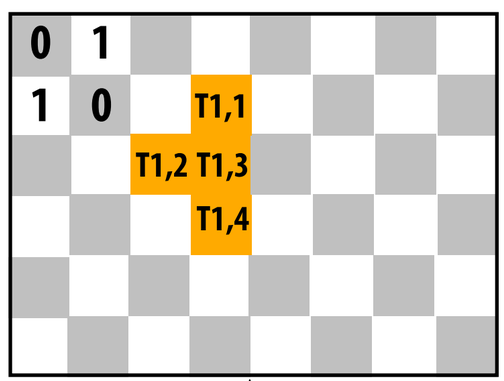
Interleaved render target

Fragments from T3 cannot be processed yet. Why?



Input:

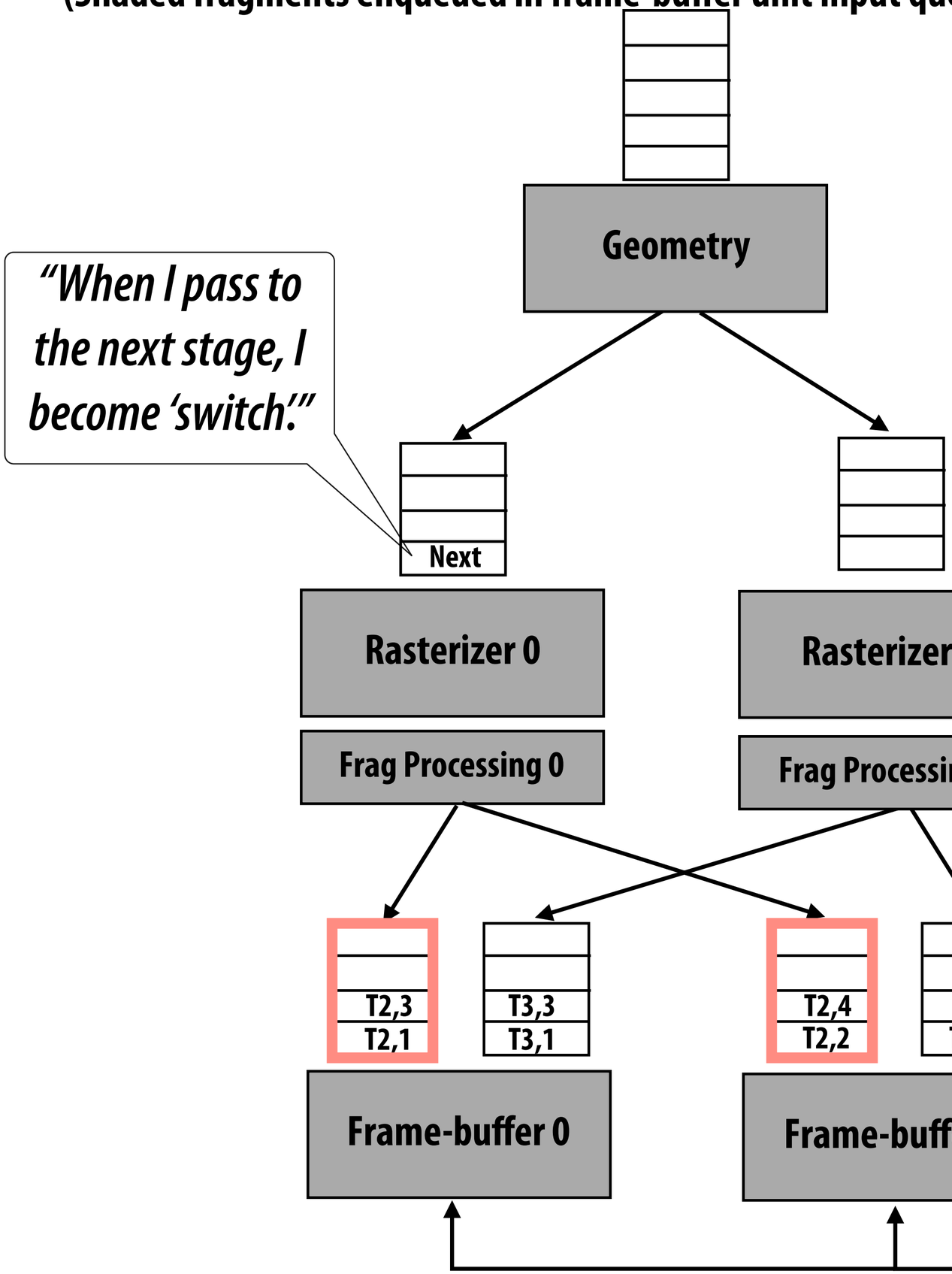




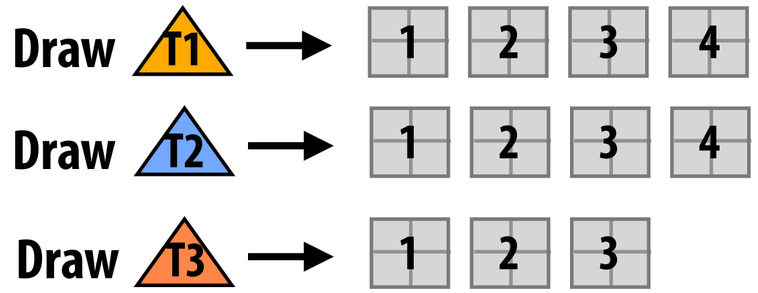
Interleaved render target

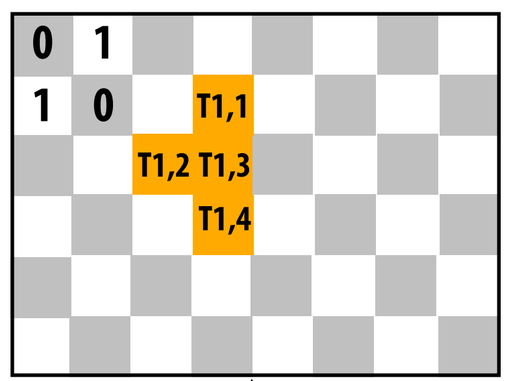
Rast 0 processes T2

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



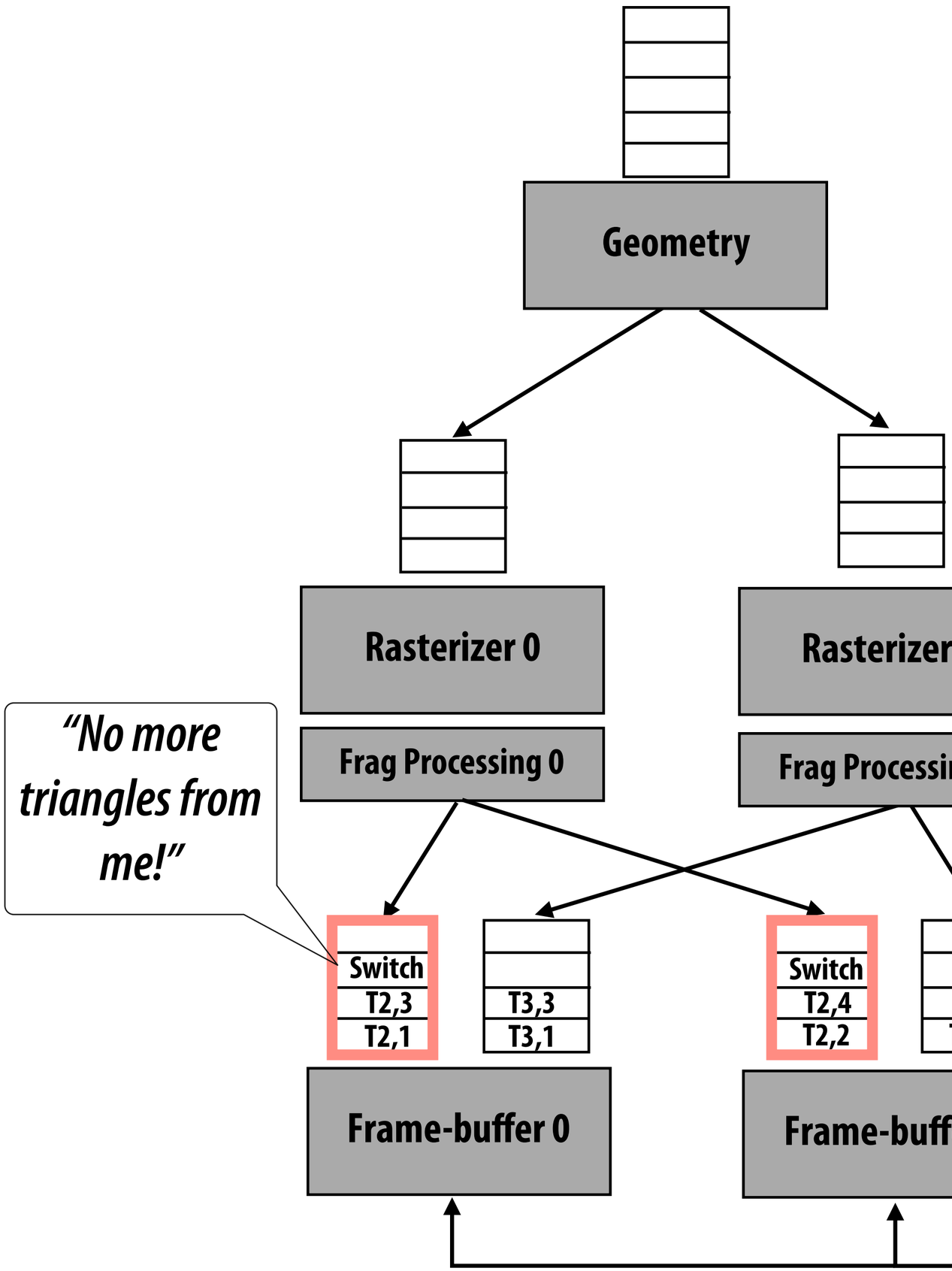
Input:



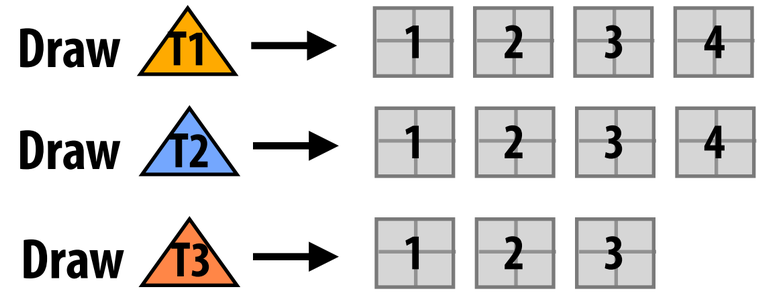


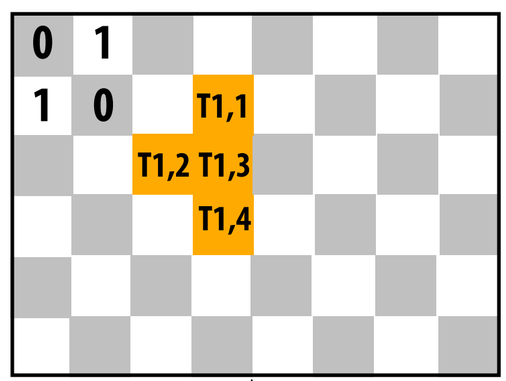
Interleaved render target

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



Input:

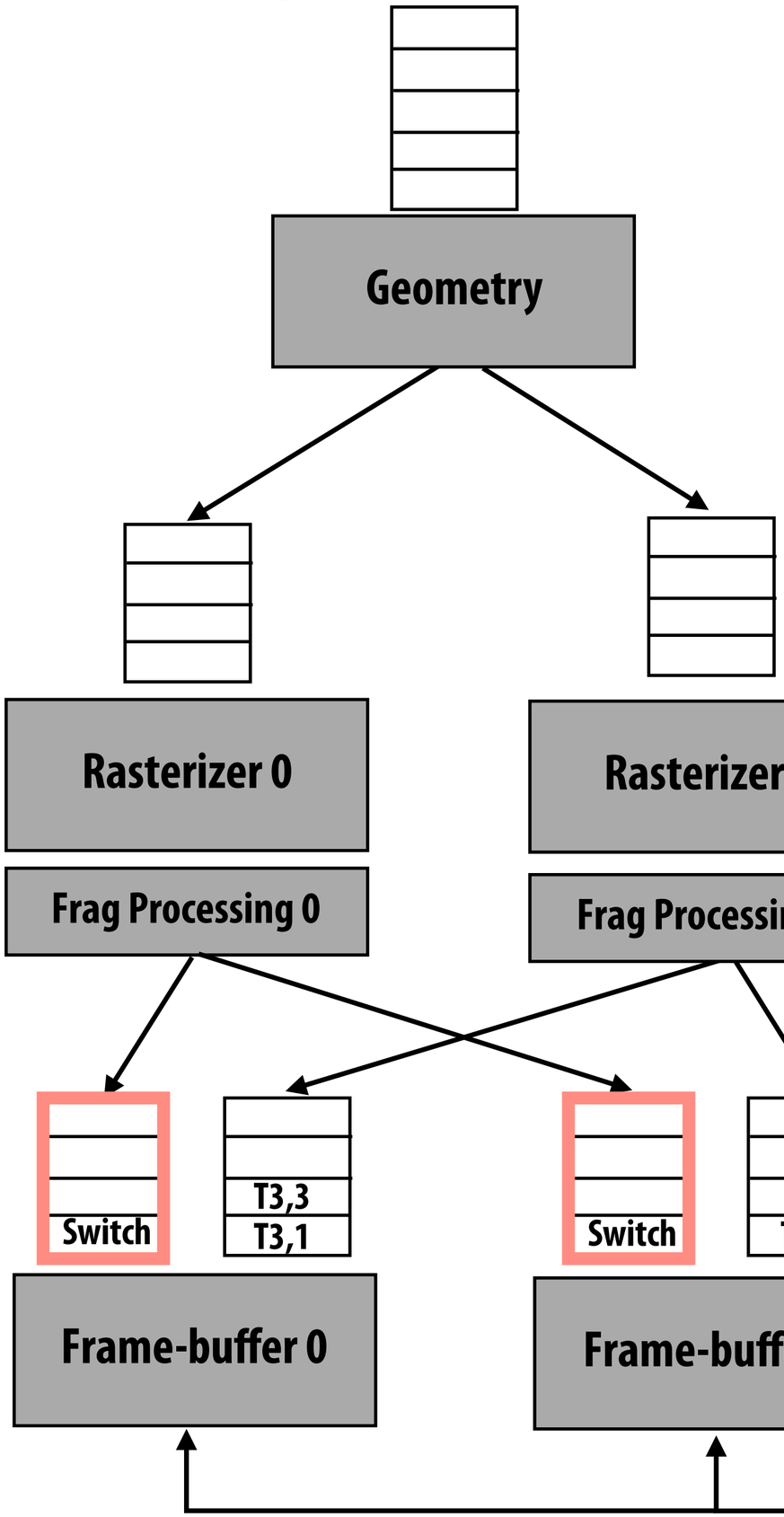




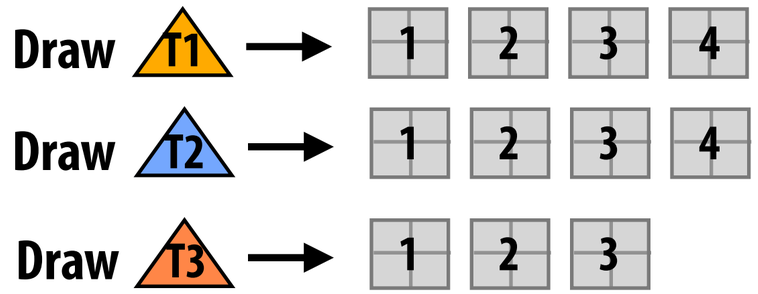
Interleaved render target

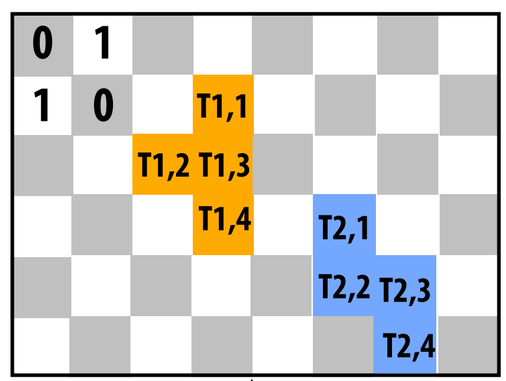
FB 0 and FB 1 can simultaneously process fragments from rast 0

(Notice updates to frame buffer)



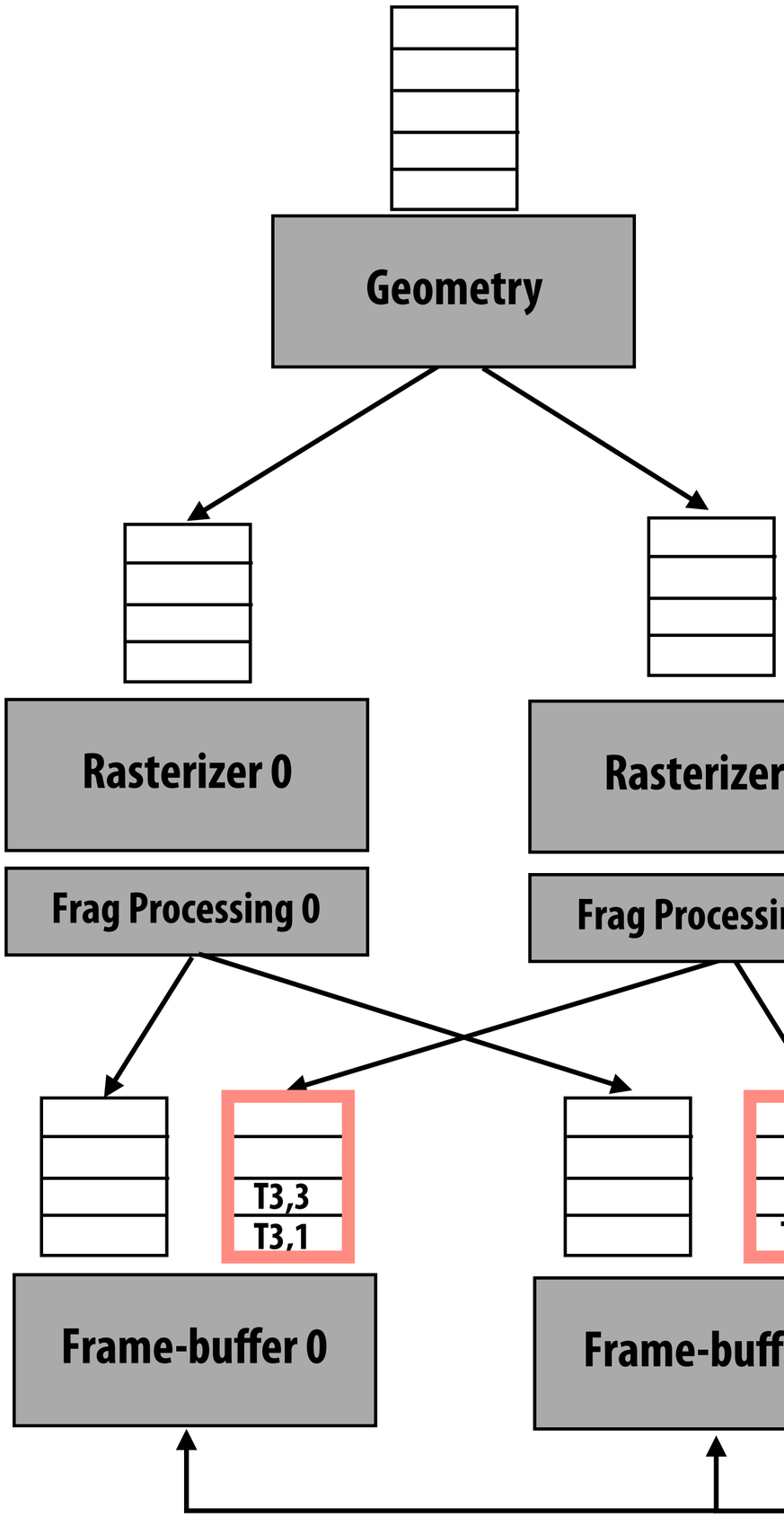
Input:



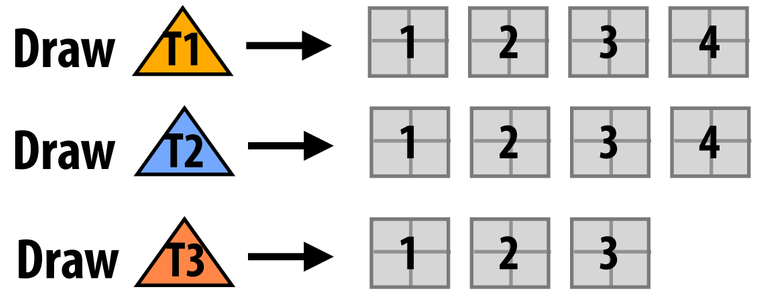


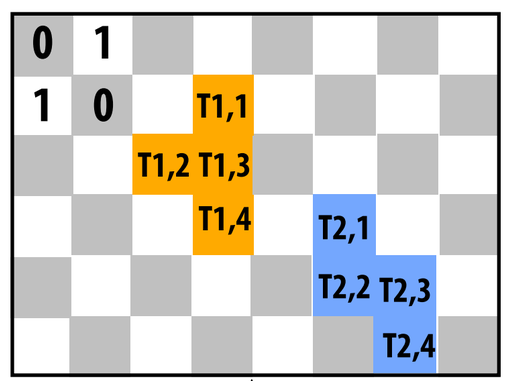
Interleaved render target

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



Input:

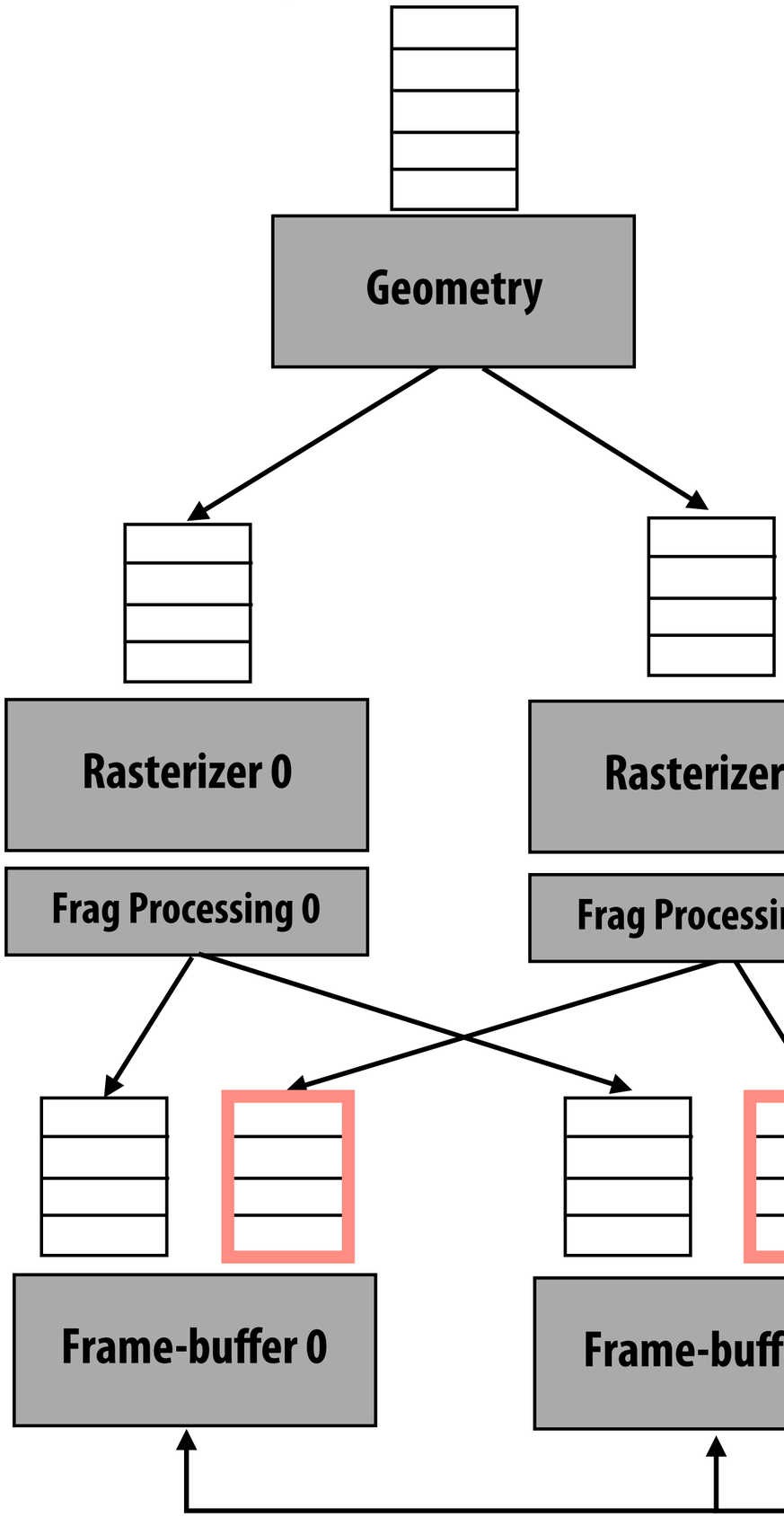




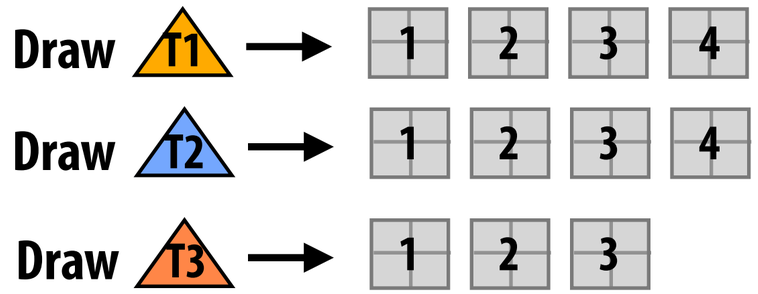
Interleaved render target

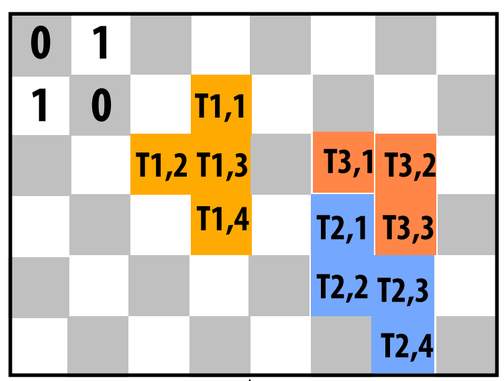
FB 0 and FB 1 can simultaneously process fragments from rast 1

(Notice updates to frame buffer)



Input:

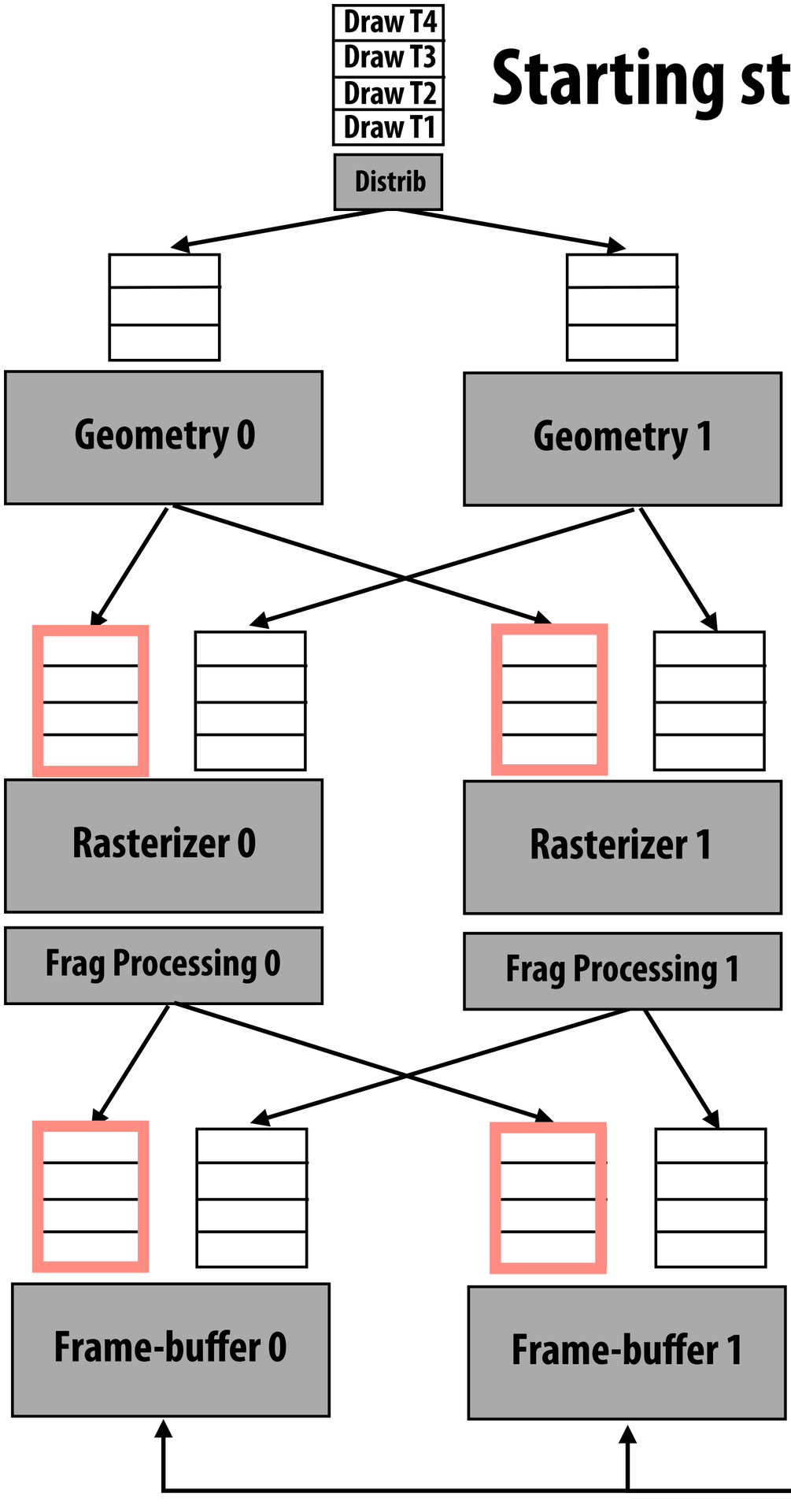




Interleaved render target

Extending to parallel geometry units

Starting state: commands enqueued



Input:

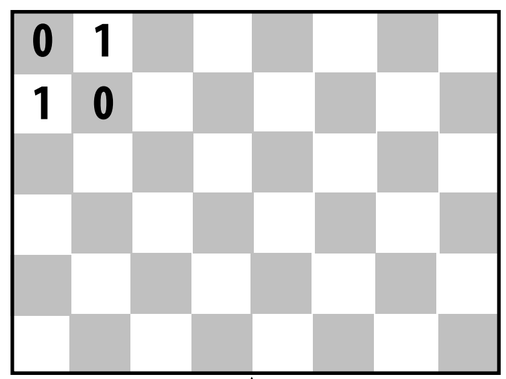
Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

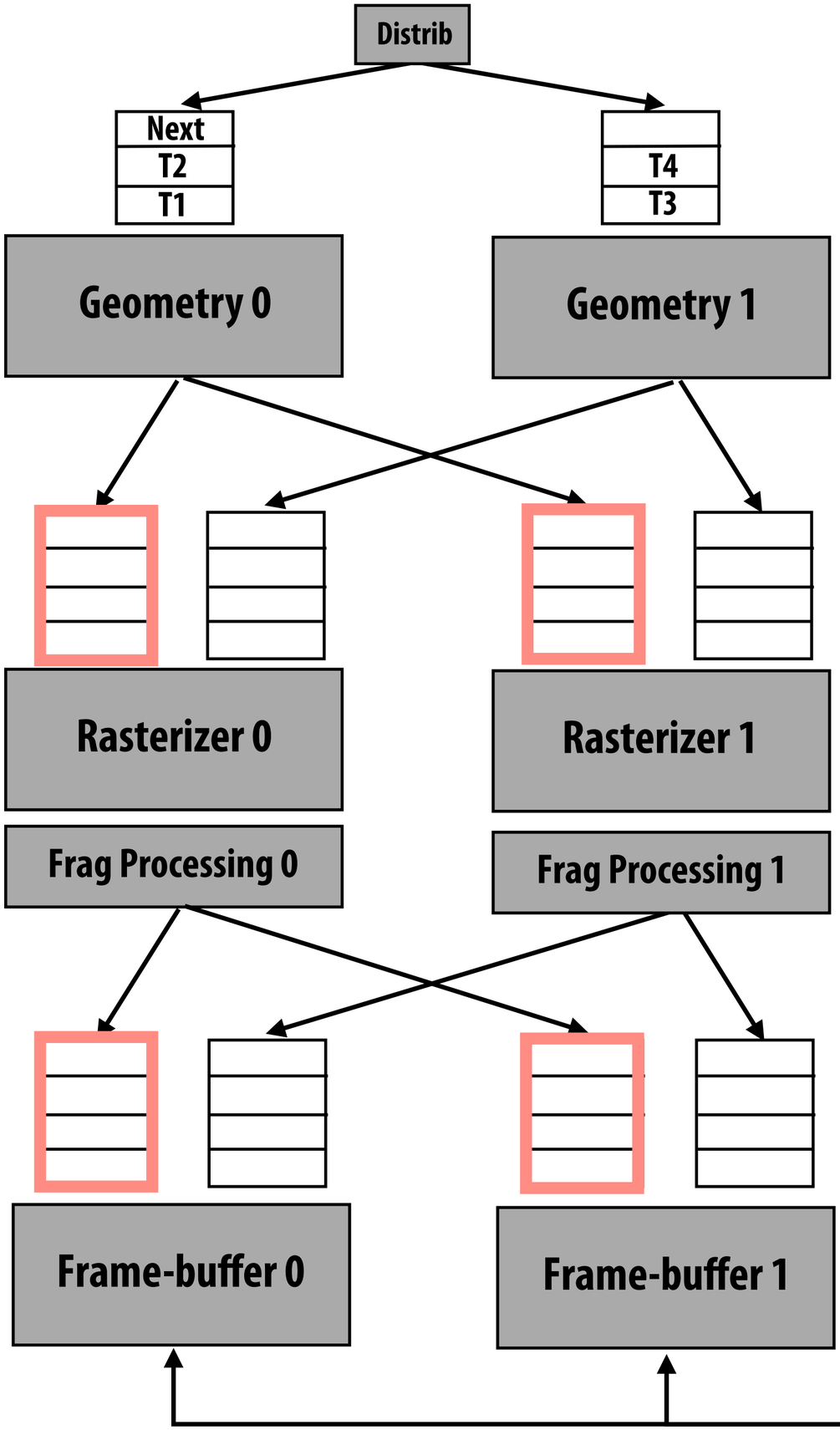
Draw T4 → 1 2

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)



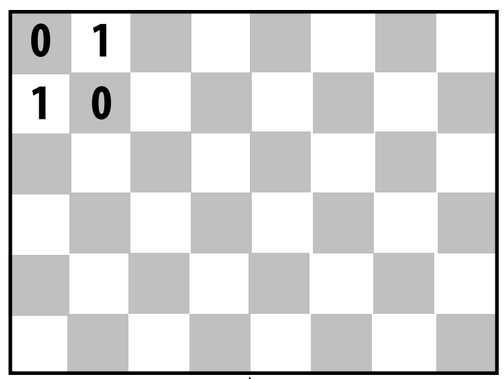
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

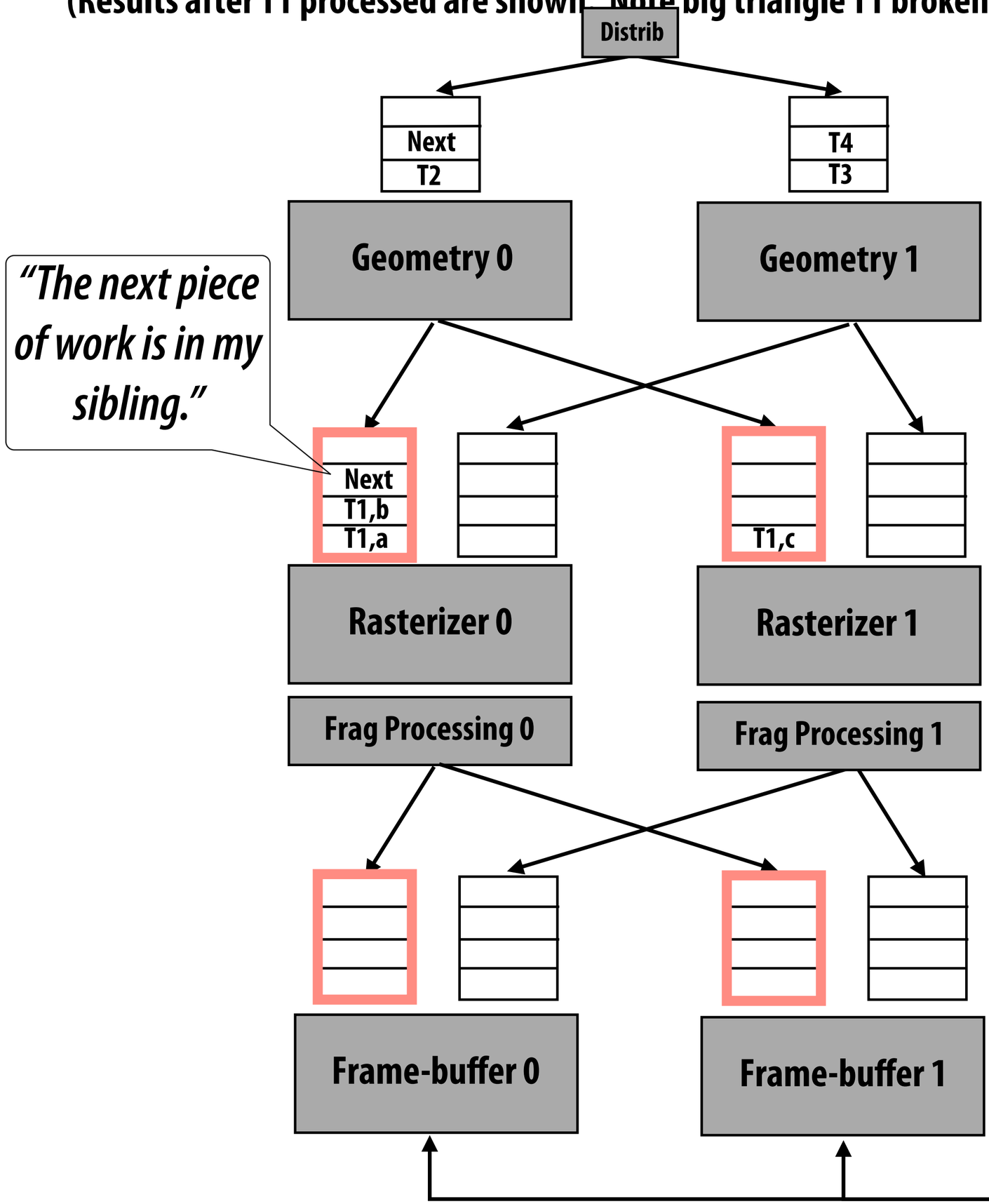
Draw T4 → 1 2



Interleaved render target

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.])



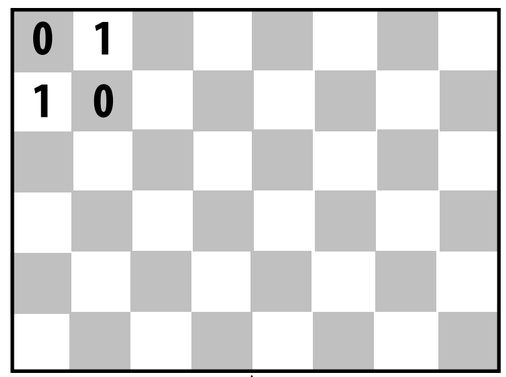
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

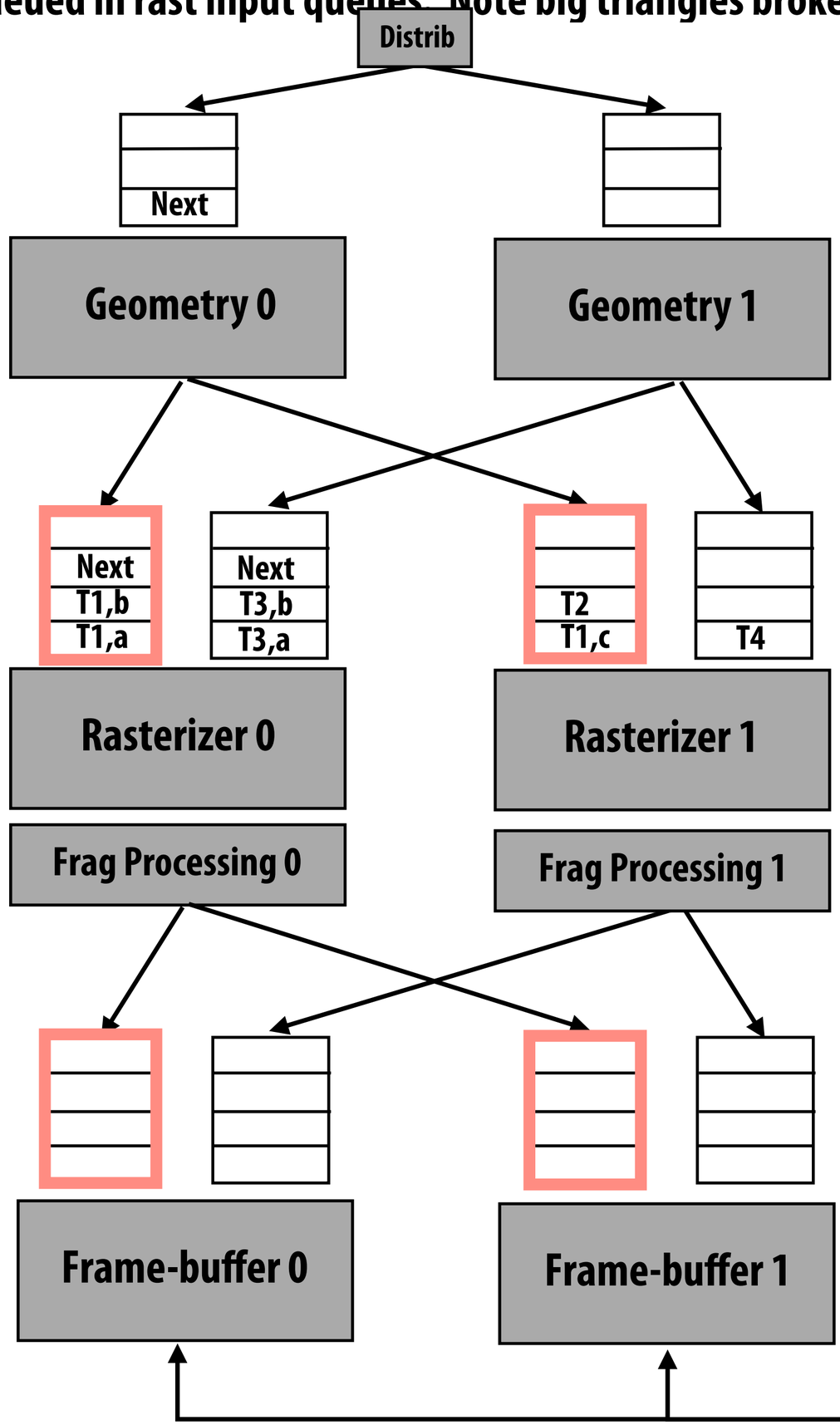
Draw T4 → 1 2



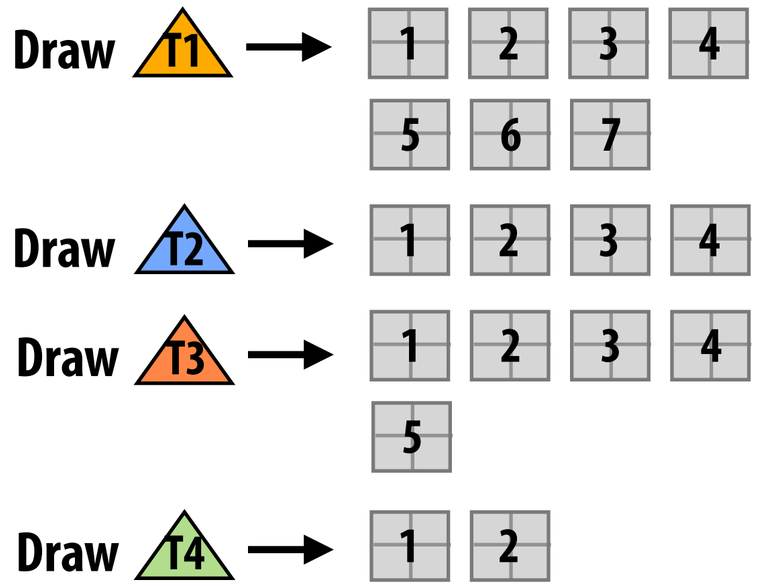
Interleaved render target

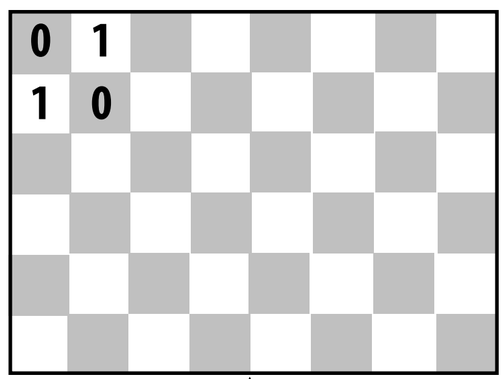
Geom 0 and geom 1 process triangles in parallel

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



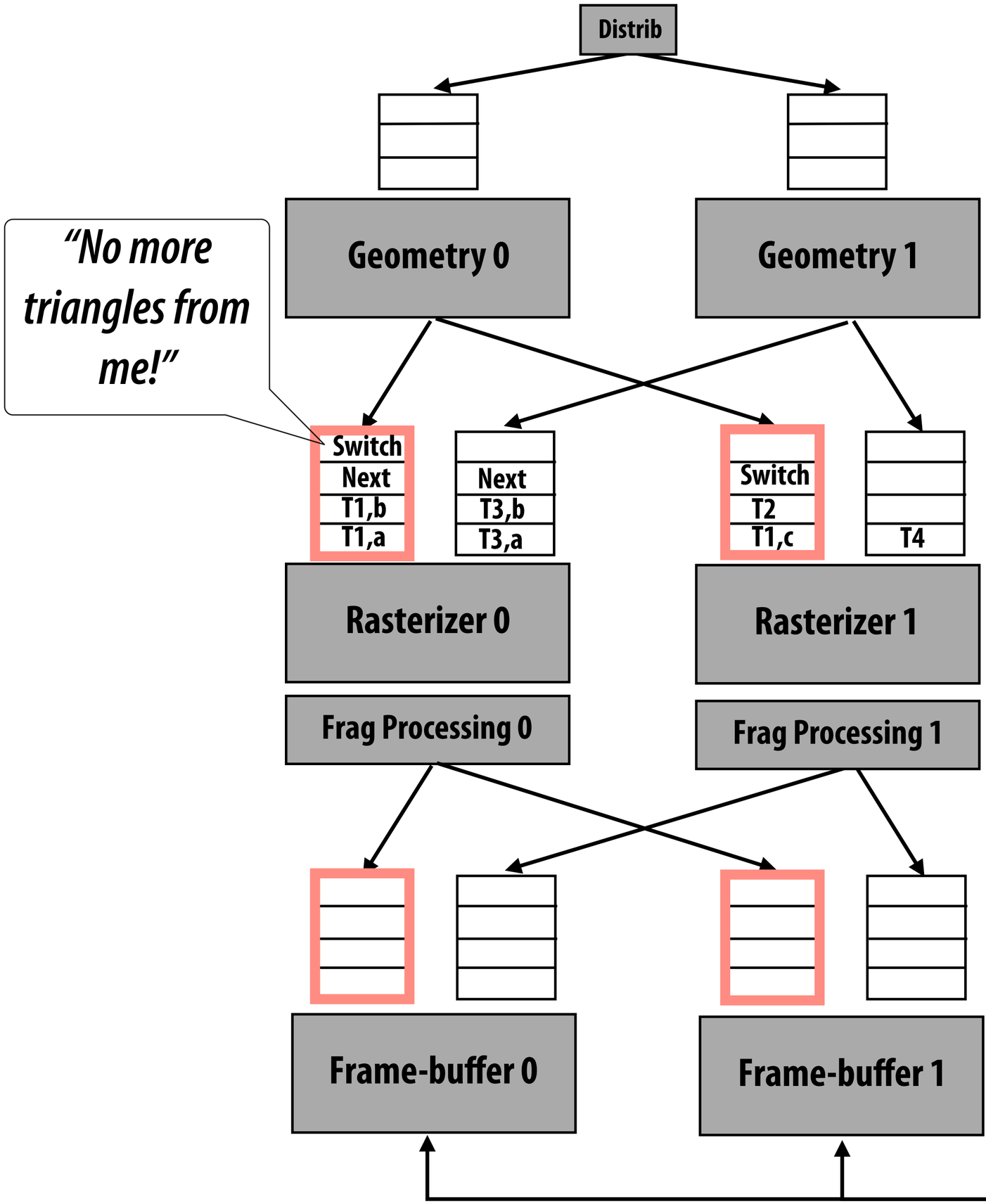
Input:





Interleaved render target

Geom 0 broadcasts 'next' token to rasterizers



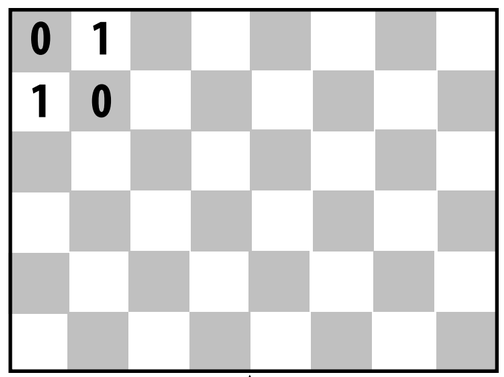
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

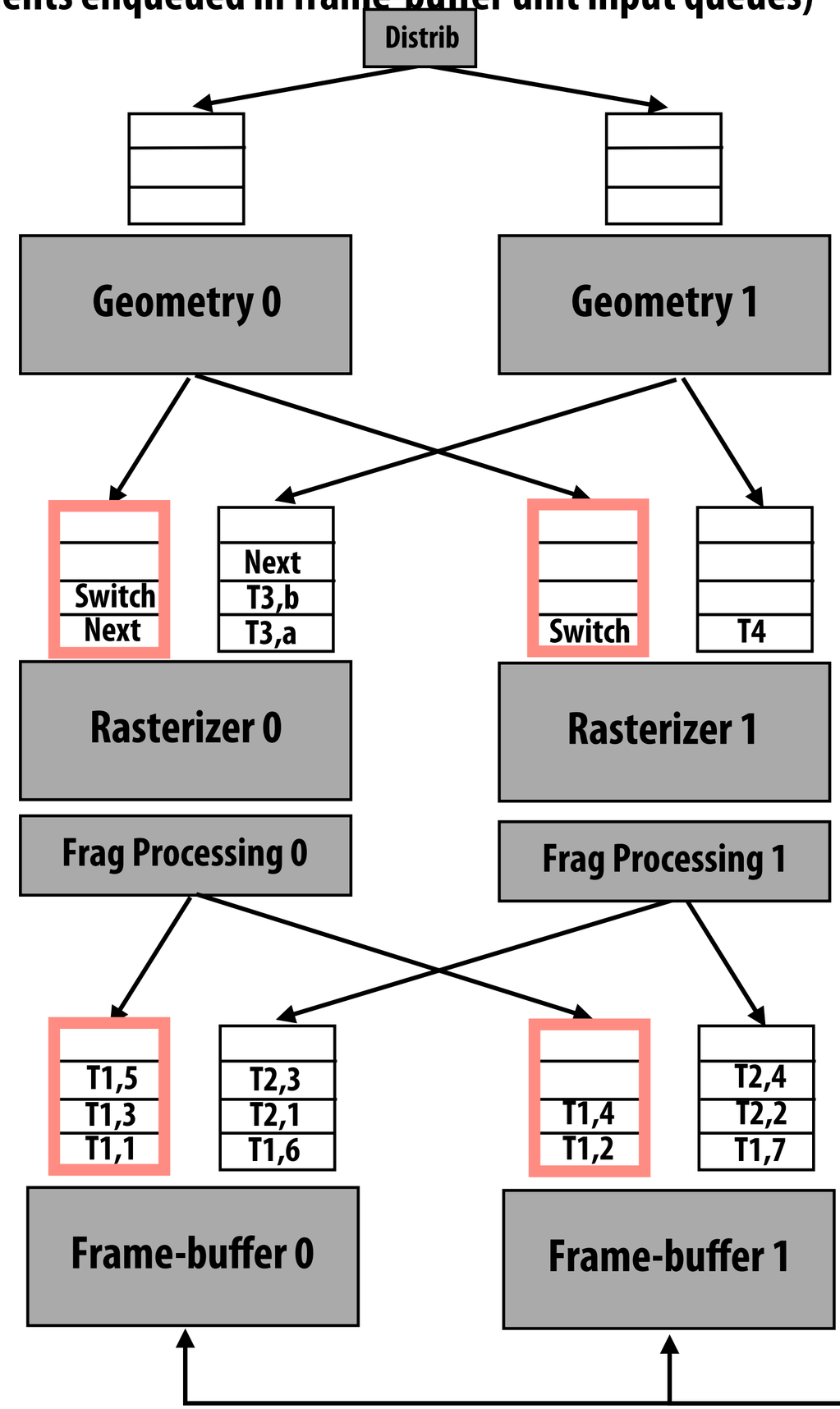
Draw T4 → 1 2



Interleaved render target

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

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



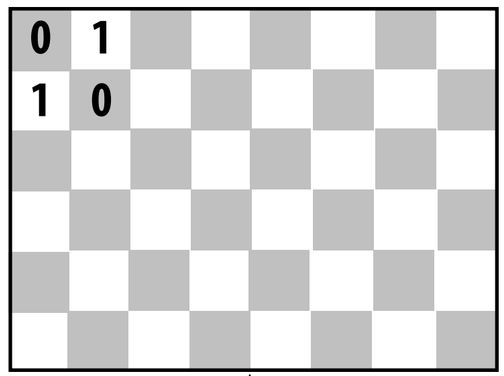
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

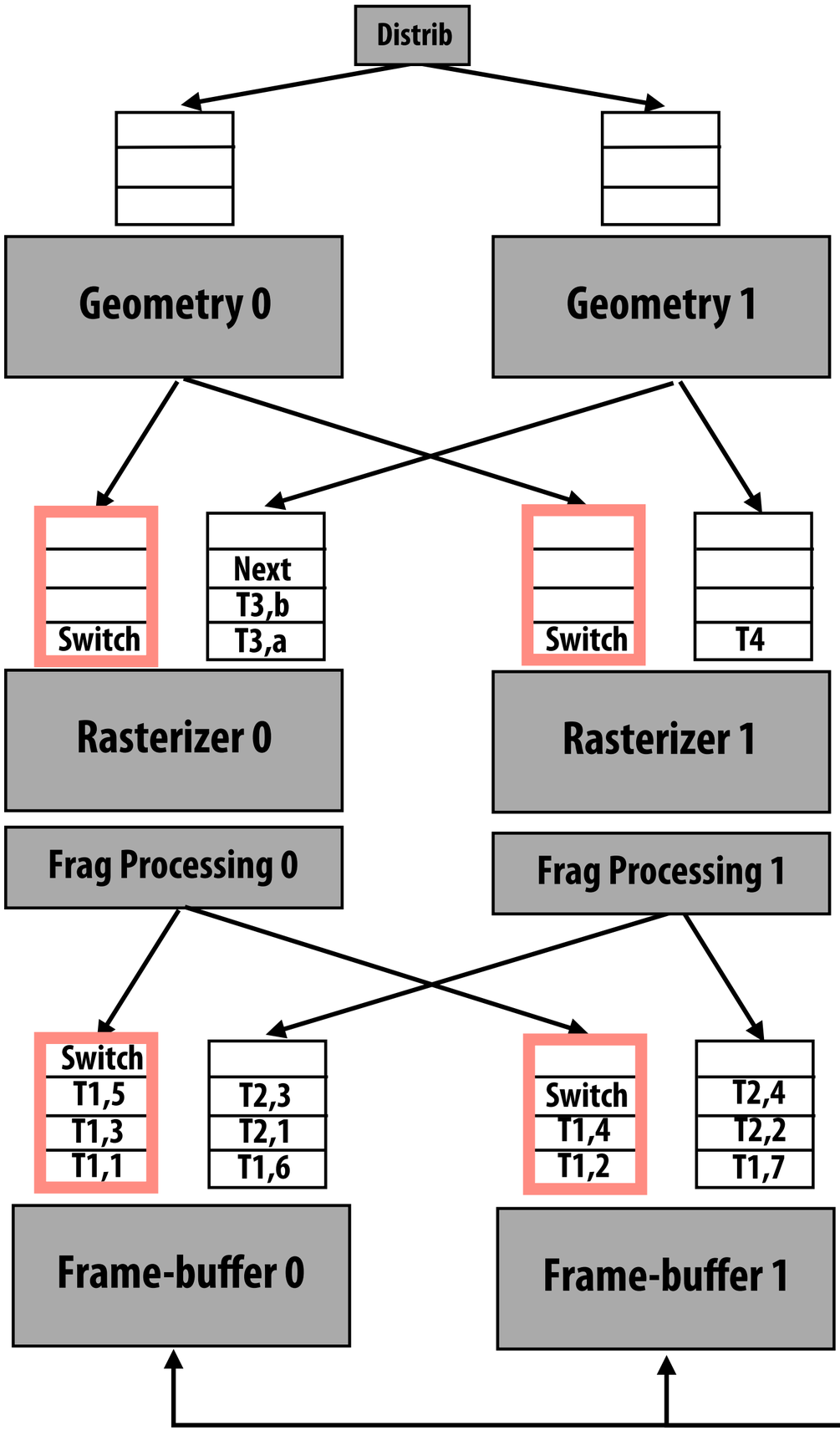
Draw T3 → 1 2 3 4 5

Draw T4 → 1 2



Interleaved render target

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



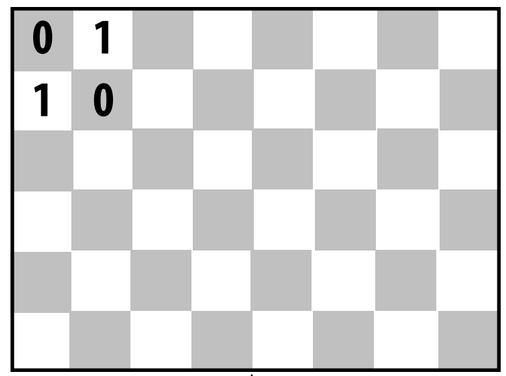
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

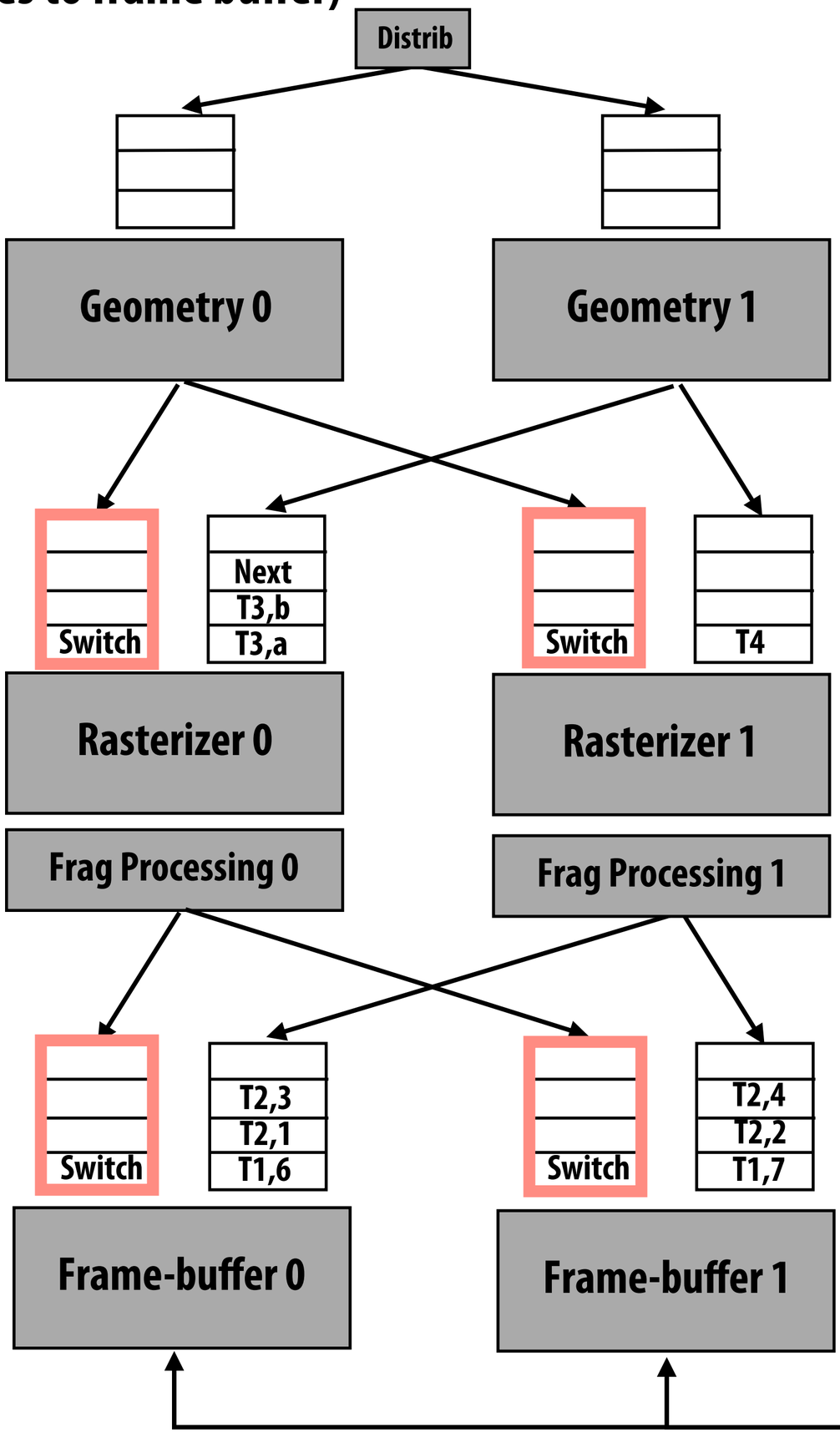
Draw T4 → 1 2



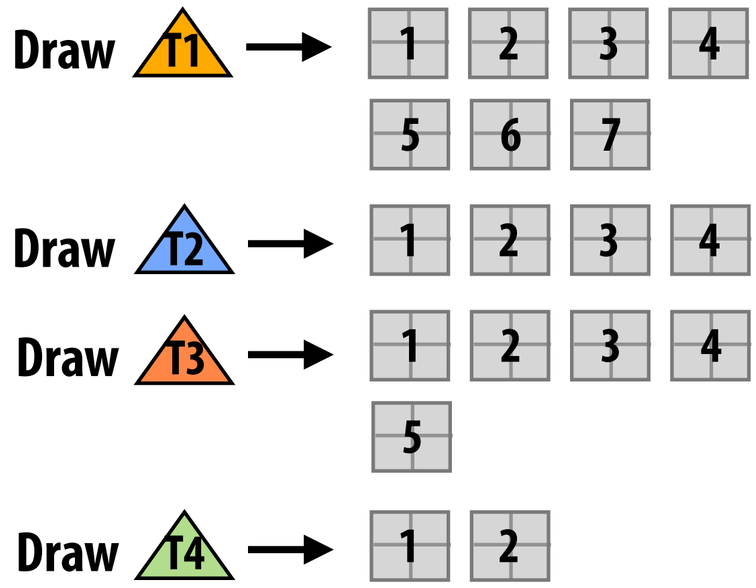
Interleaved render target

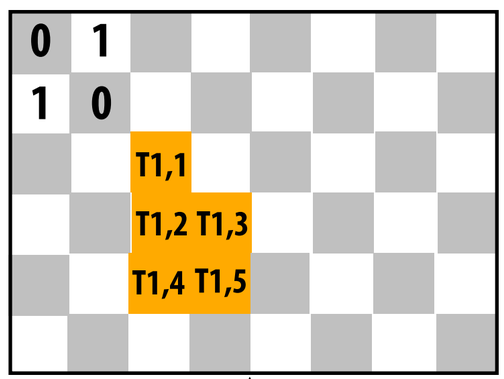
Frame-buffer units process frags from (geom 0, rast 0) in parallel

(Notice updates to frame buffer)



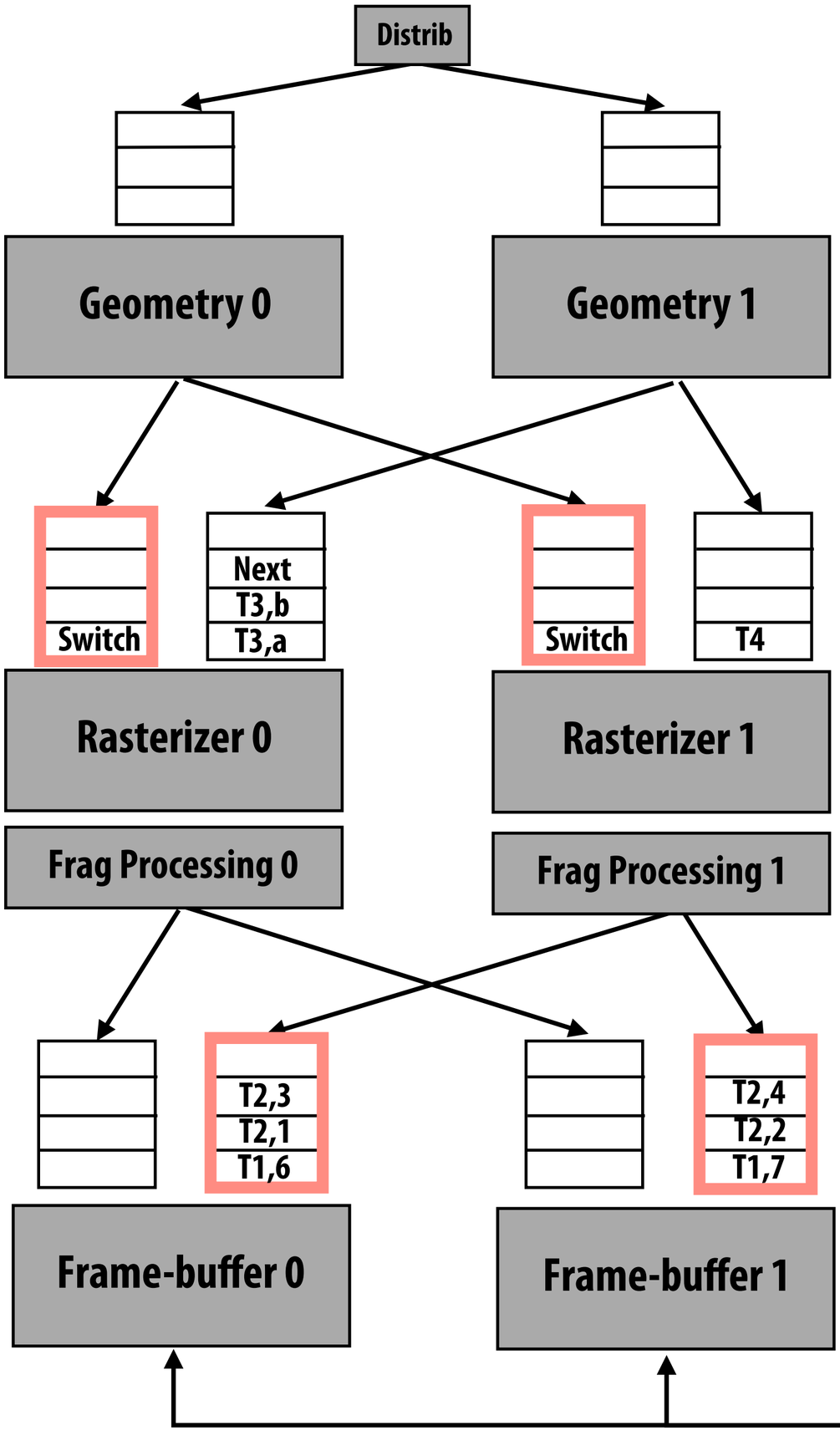
Input:



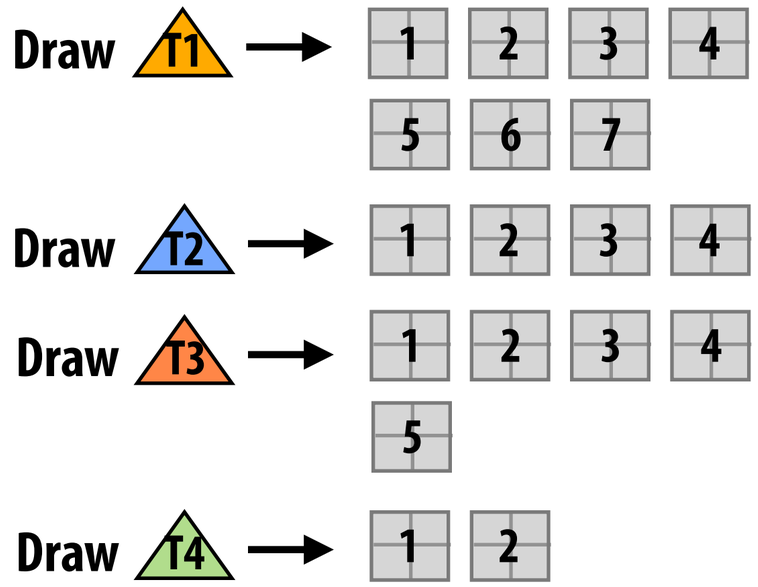


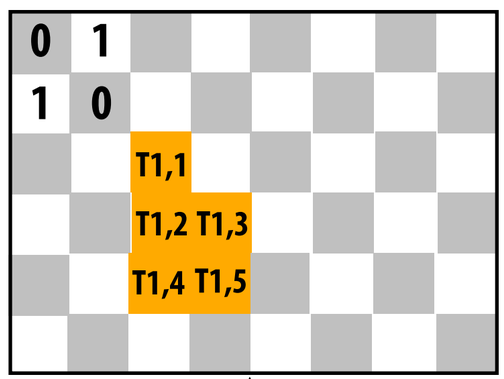
Interleaved render target

“End of rast 0” token reached by FB: FB units start processing input from rast 1 (fragments from geom 0, rast 1)



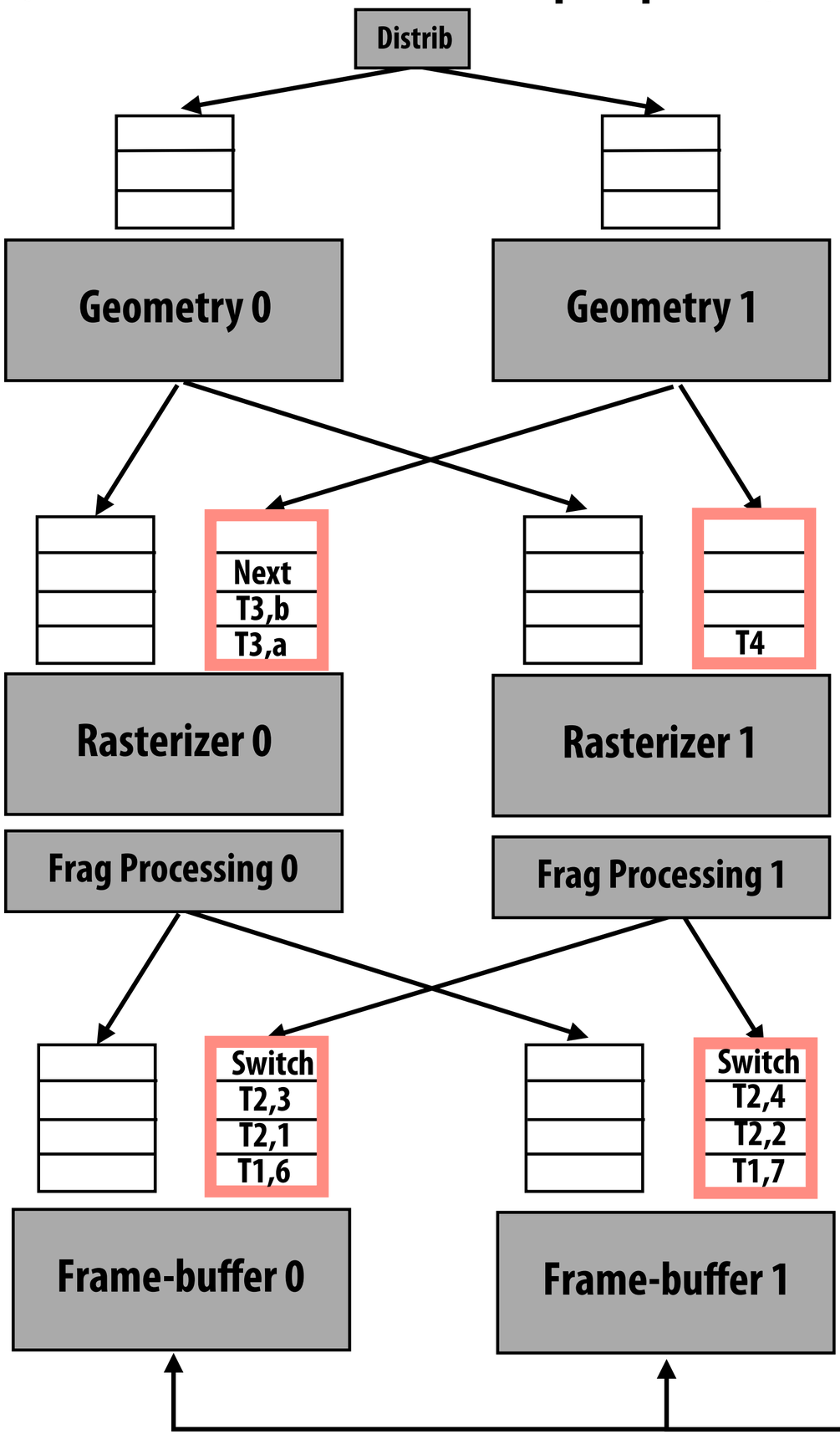
Input:





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)



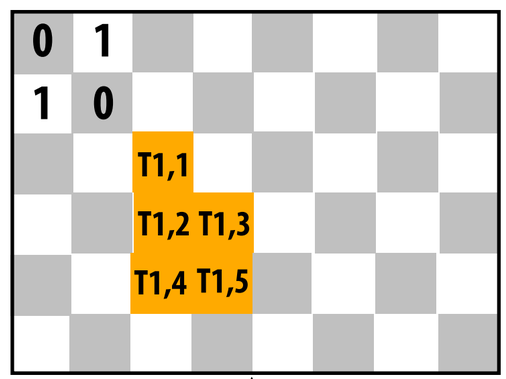
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

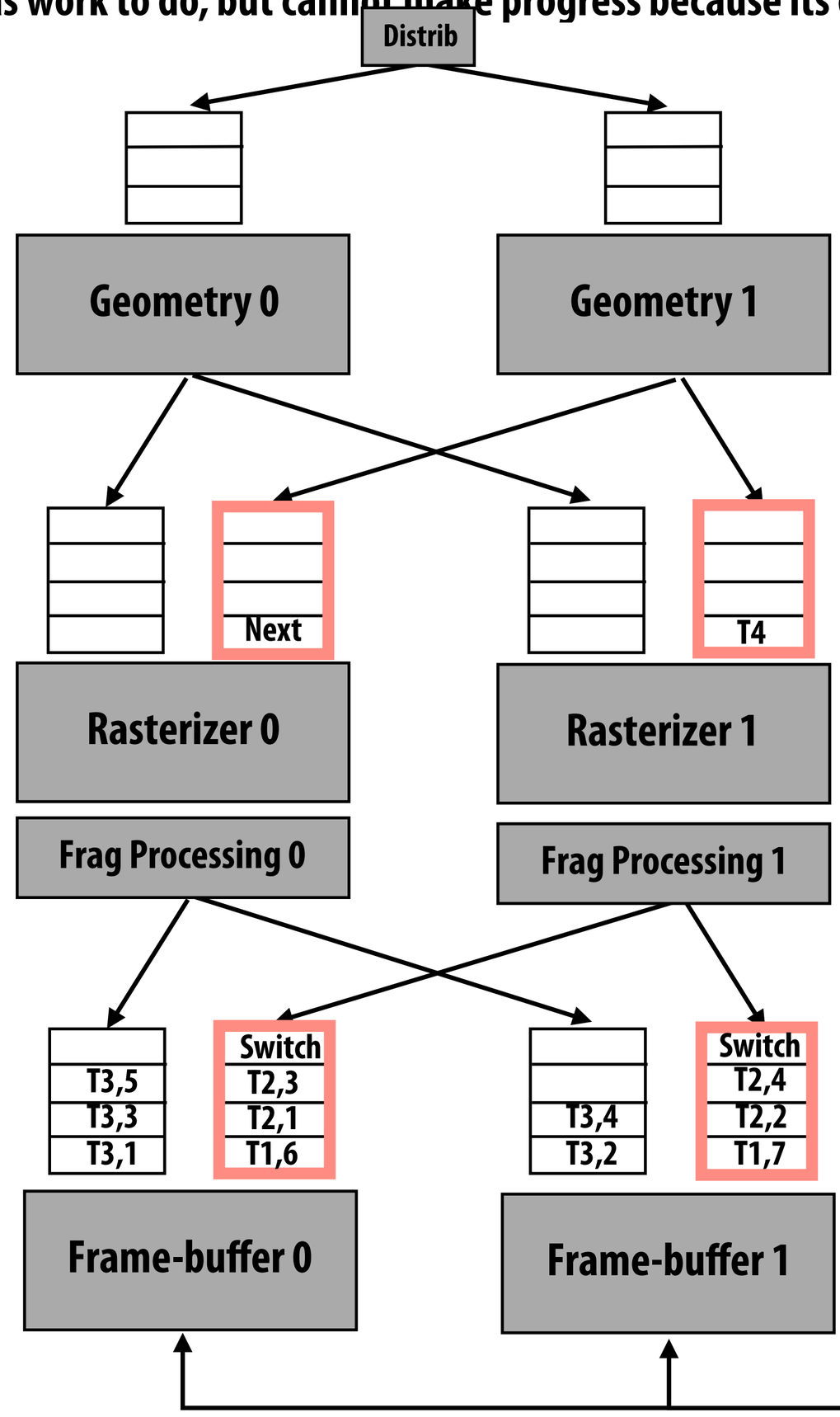
Draw T4 → 1 2



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)



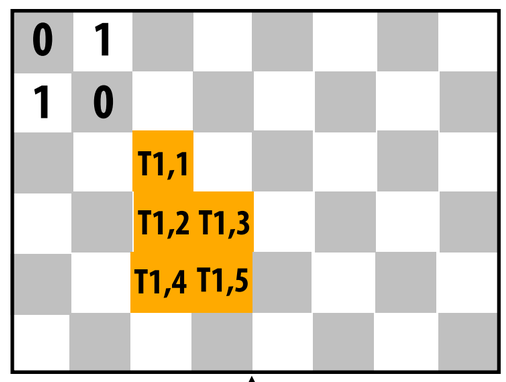
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

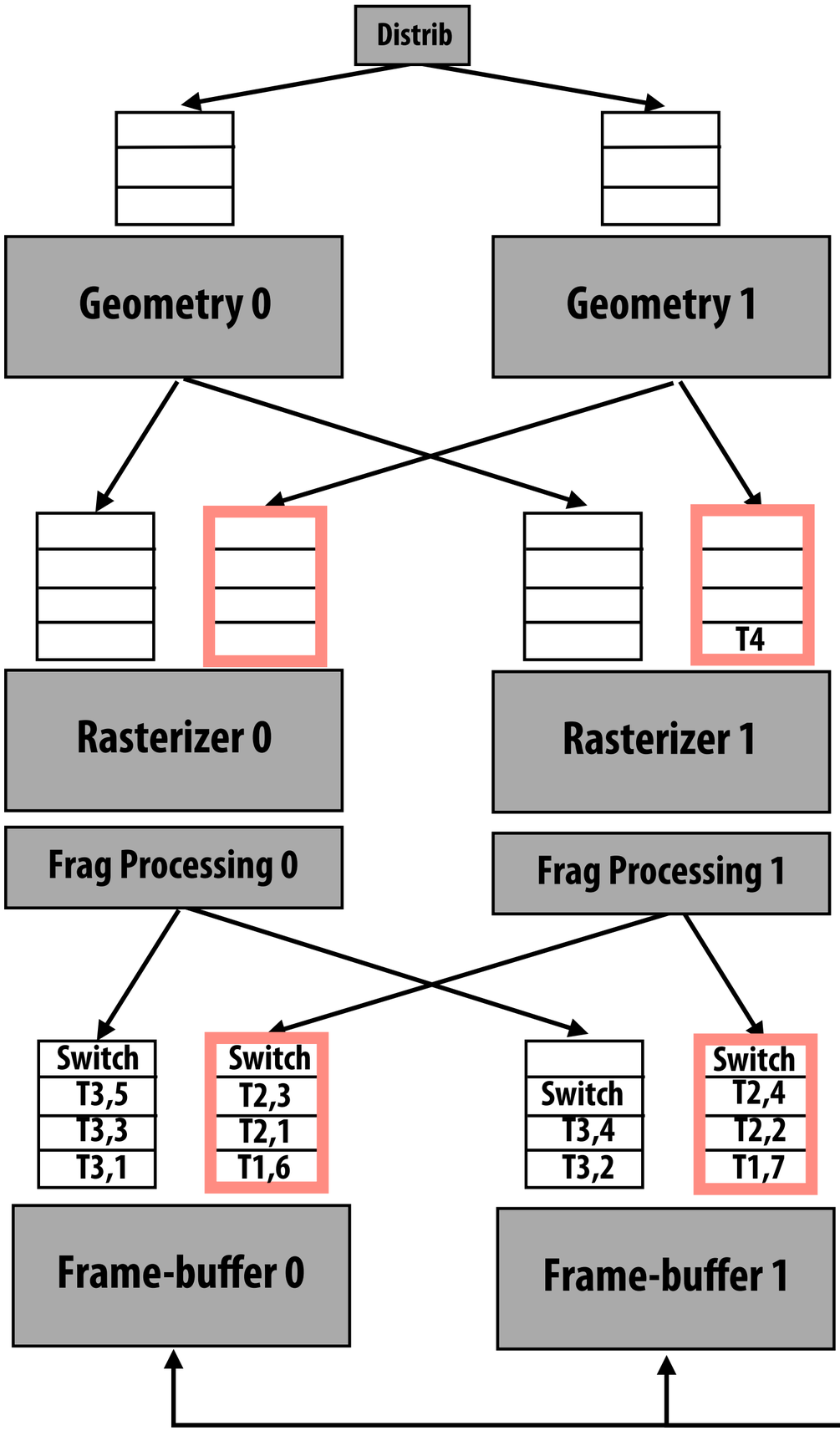
Draw T3 → 1 2 3 4 5

Draw T4 → 1 2



Interleaved render target

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



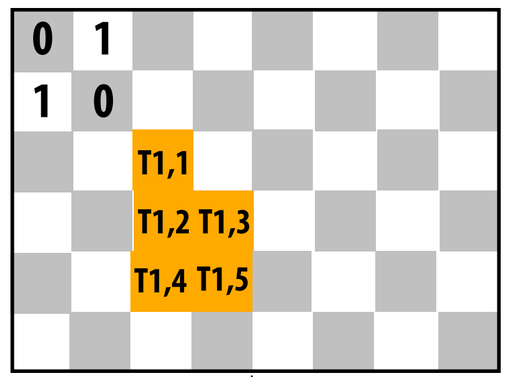
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

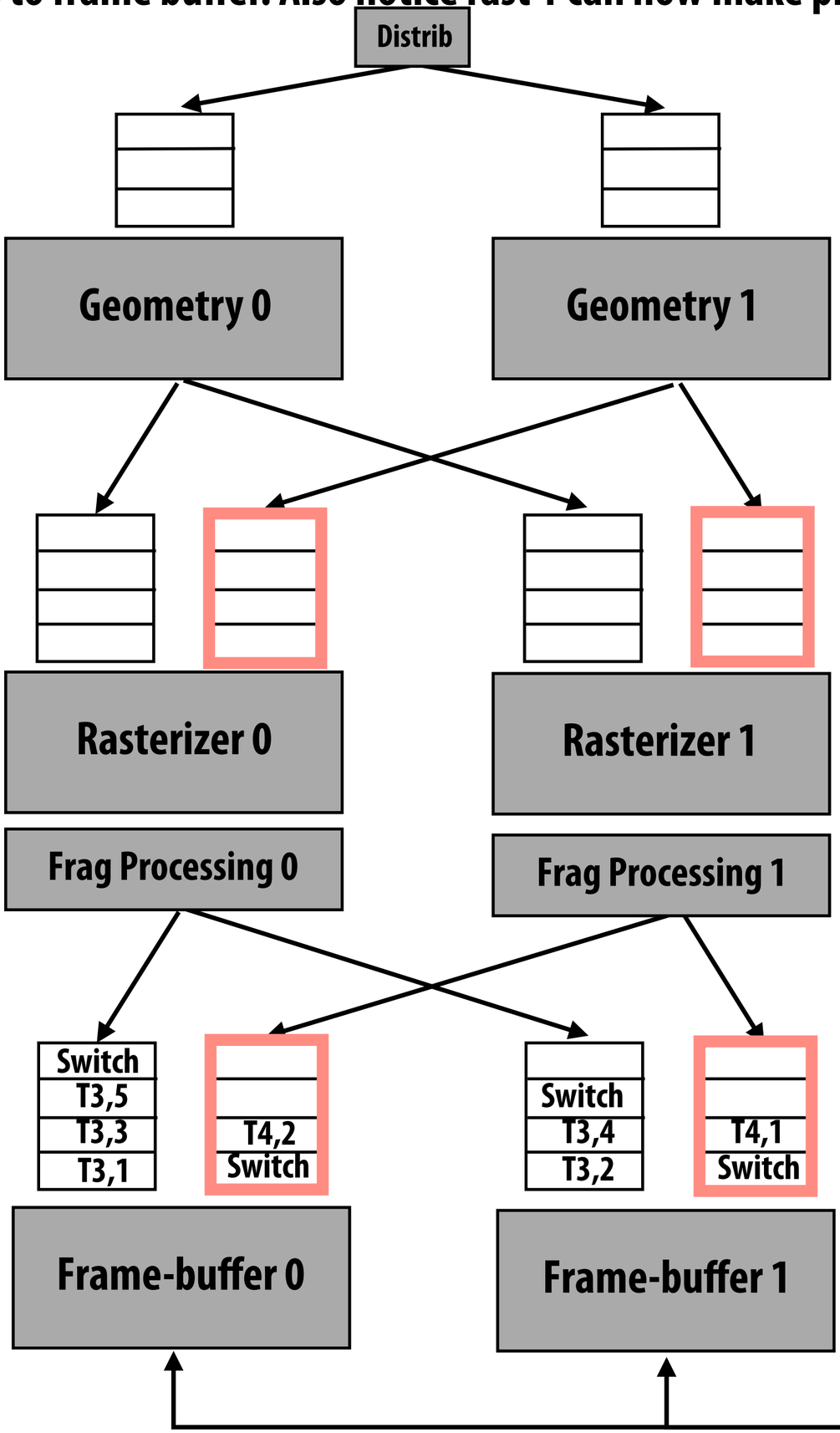
Draw T4 → 1 2



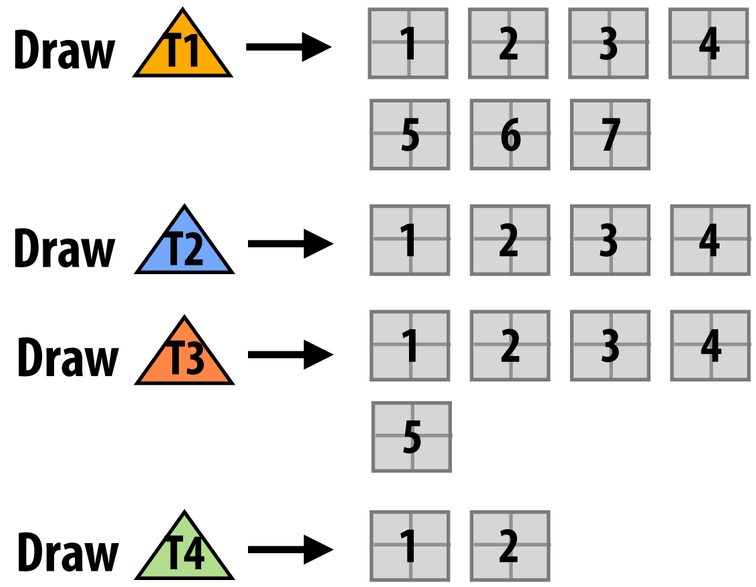
Interleaved render target

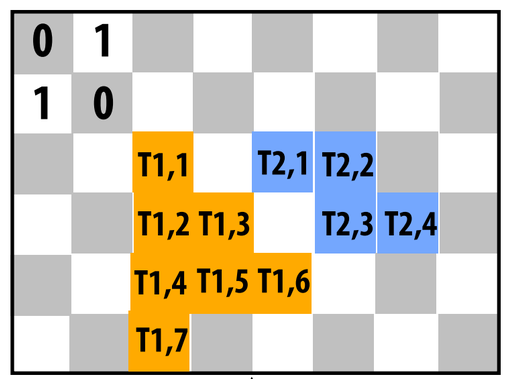
Frame-buffer units process frags from (geom 0, rast 1) in parallel

(Notice updates to frame buffer. Also notice rast 1 can now make progress since space has become available)



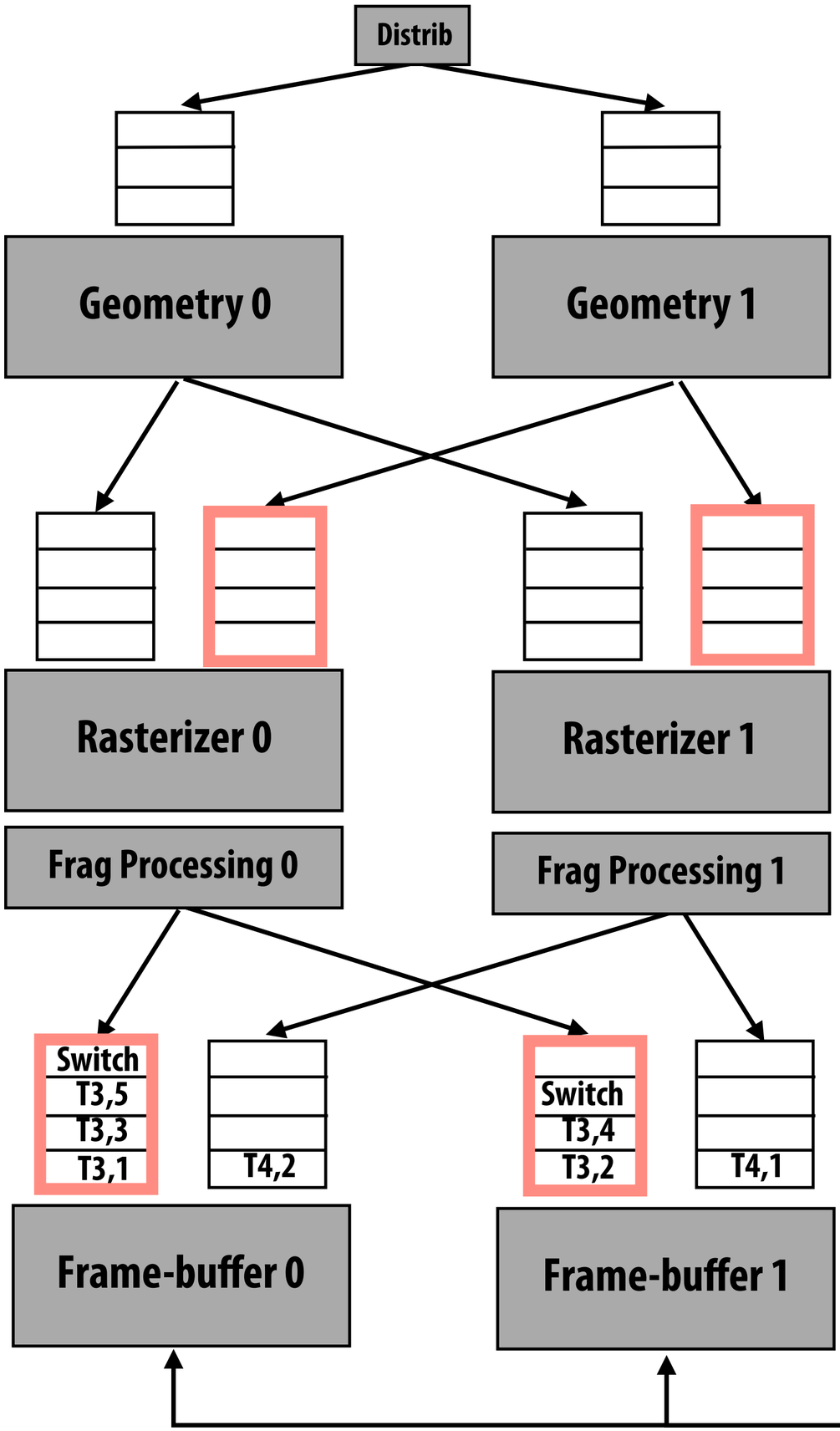
Input:



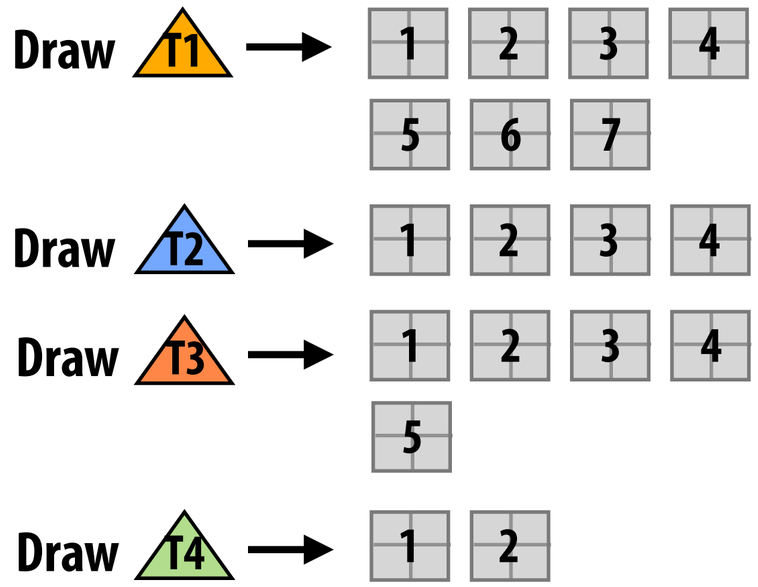


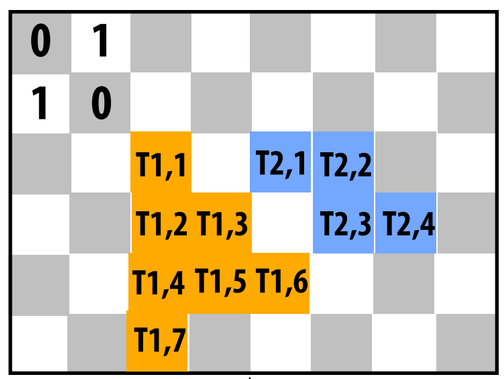
Interleaved render target

Switch token reached by FB: FB units start processing input from (geom 1, rast 0)



Input:

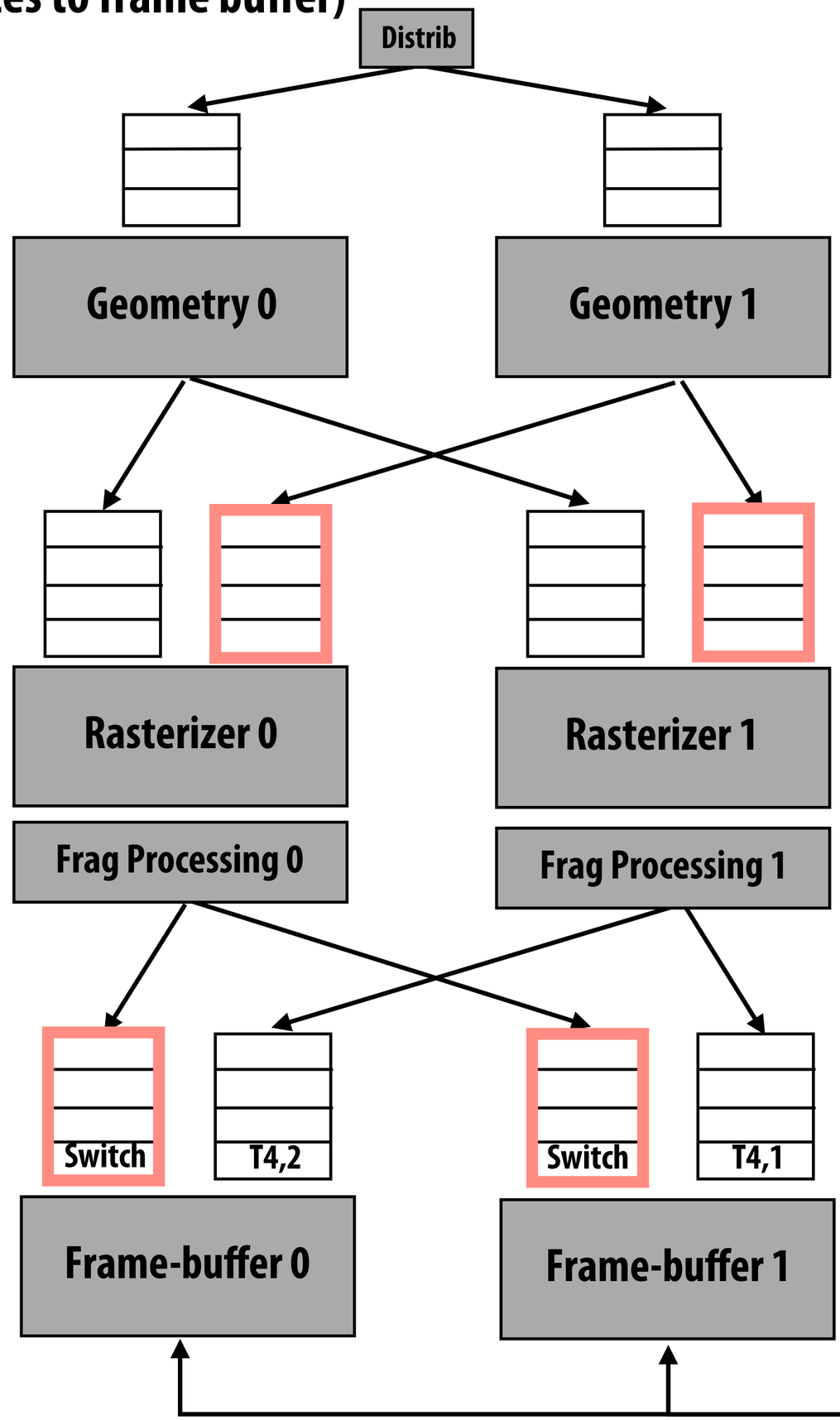




Interleaved render target

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

(Notice updates to frame buffer)



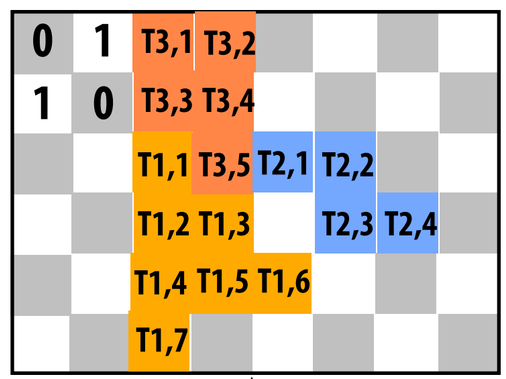
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

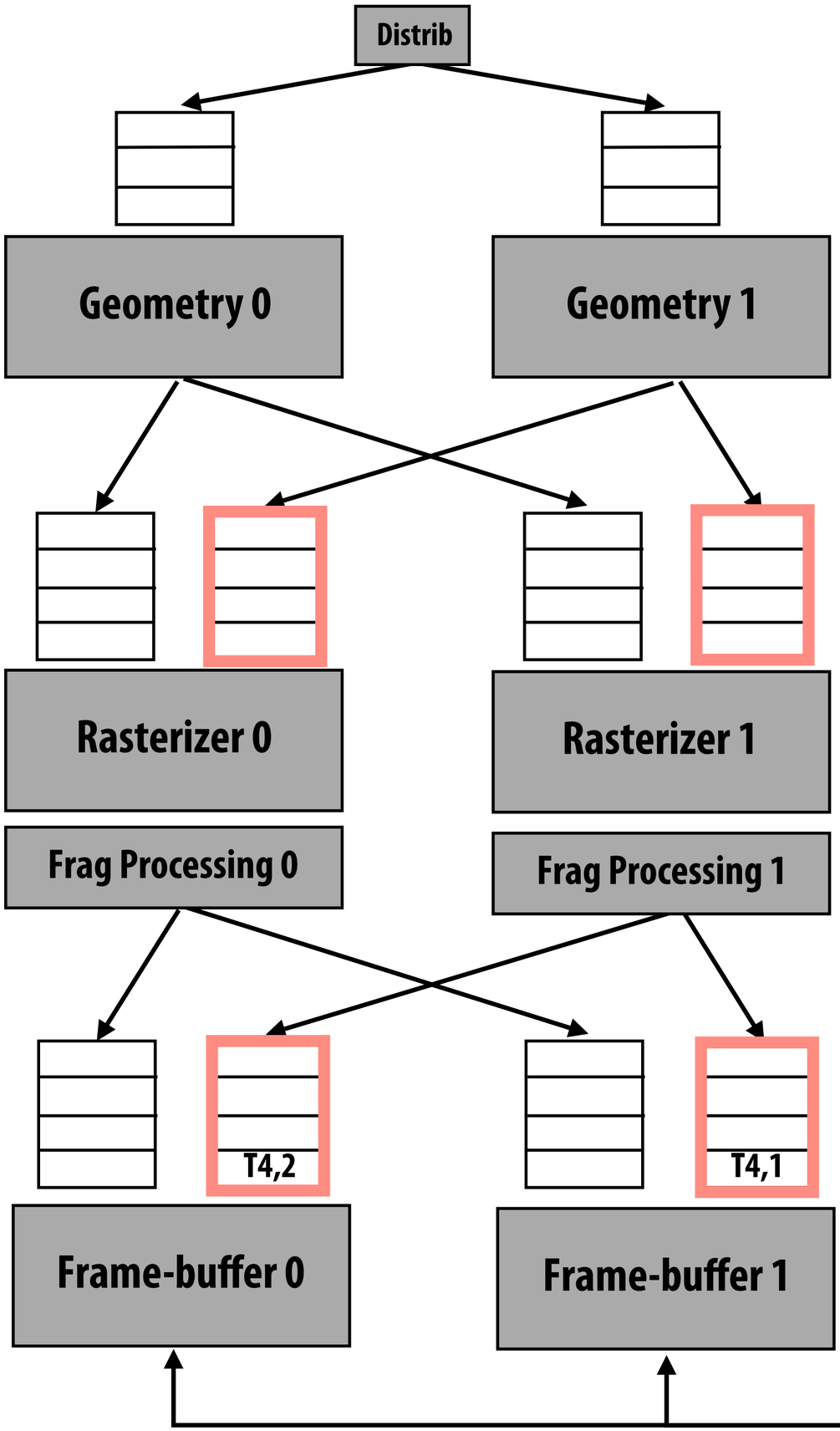
Draw T3 → 1 2 3 4 5

Draw T4 → 1 2



Interleaved render target

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



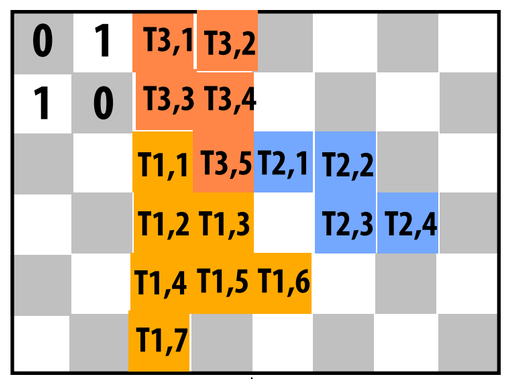
Input:

Draw T1 → 1 2 3 4 5 6 7

Draw T2 → 1 2 3 4

Draw T3 → 1 2 3 4 5

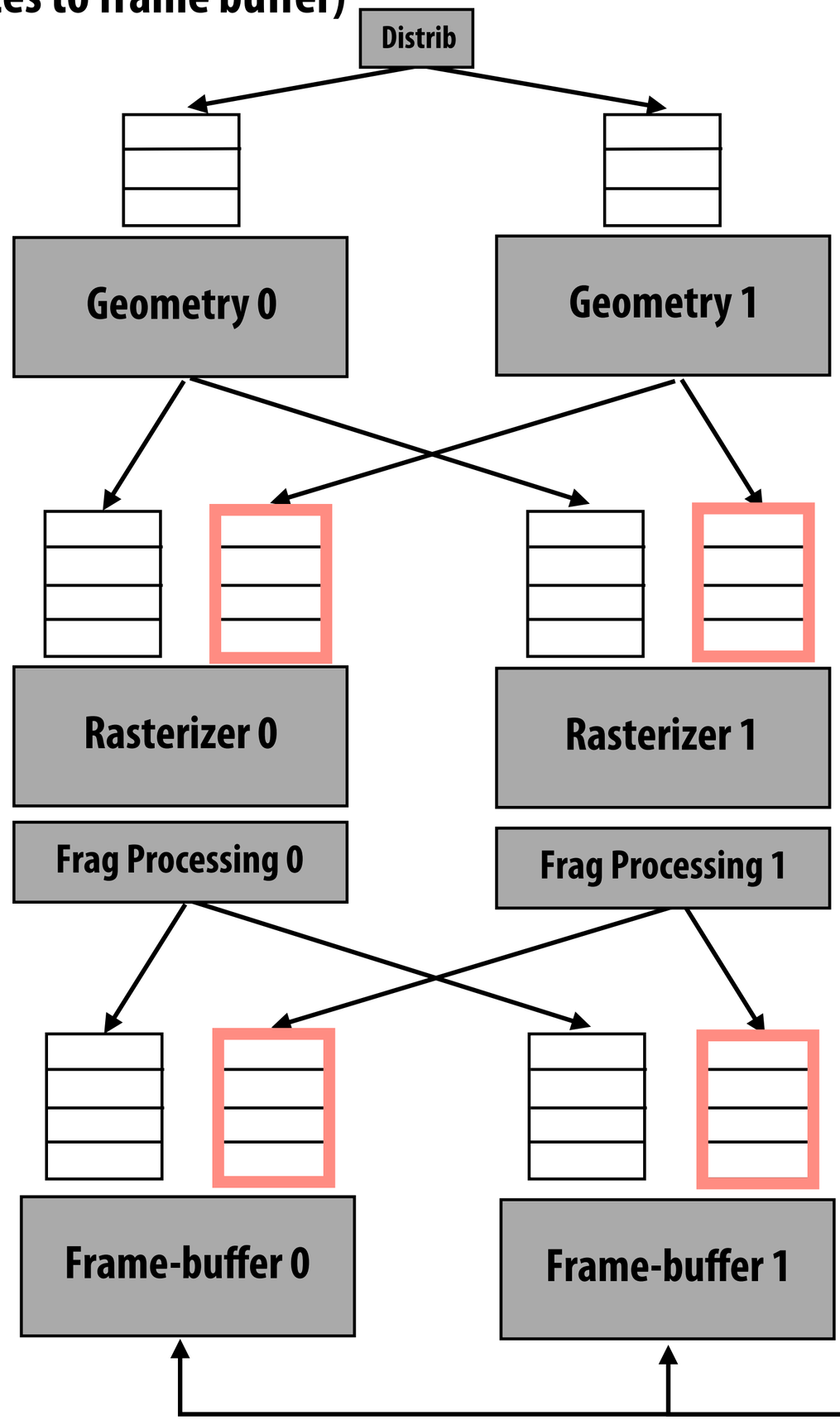
Draw T4 → 1 2



Interleaved render target

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

(Notice updates to frame buffer)



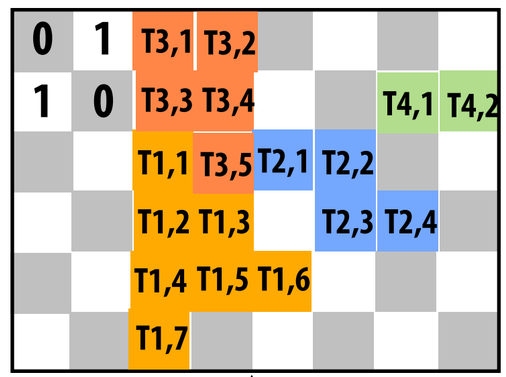
Input:

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



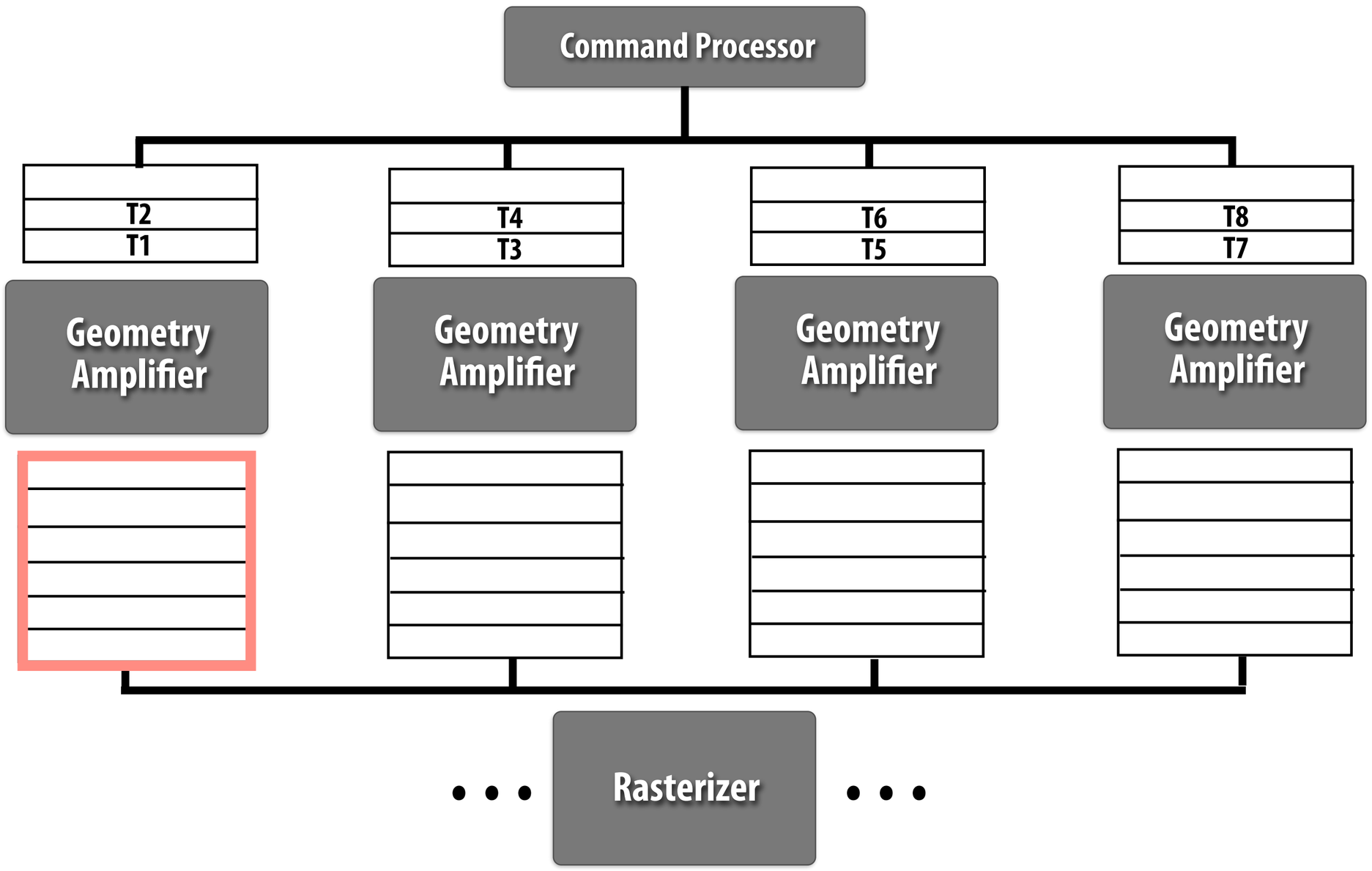
Interleaved render target

Parallel scheduling with data amplification

Geometry amplification

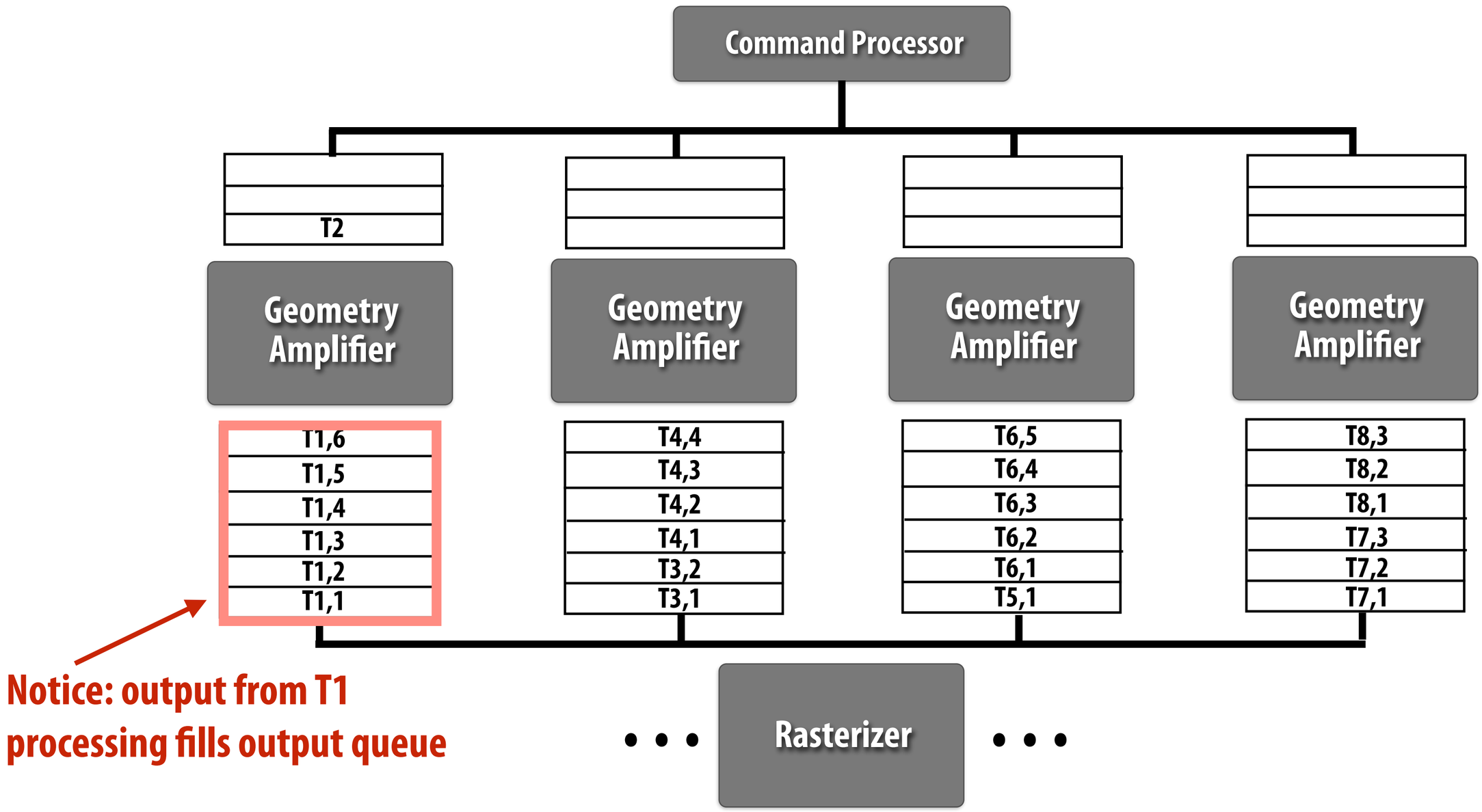
- Consider examples of one-to-many stage behavior during geometry processing in the graphics pipeline:
  - Clipping amplifies geometry (clipping can result in multiple output primitives)
  - Tessellation: pipeline permits thousands of vertices to be generated from a single base primitive (challenging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

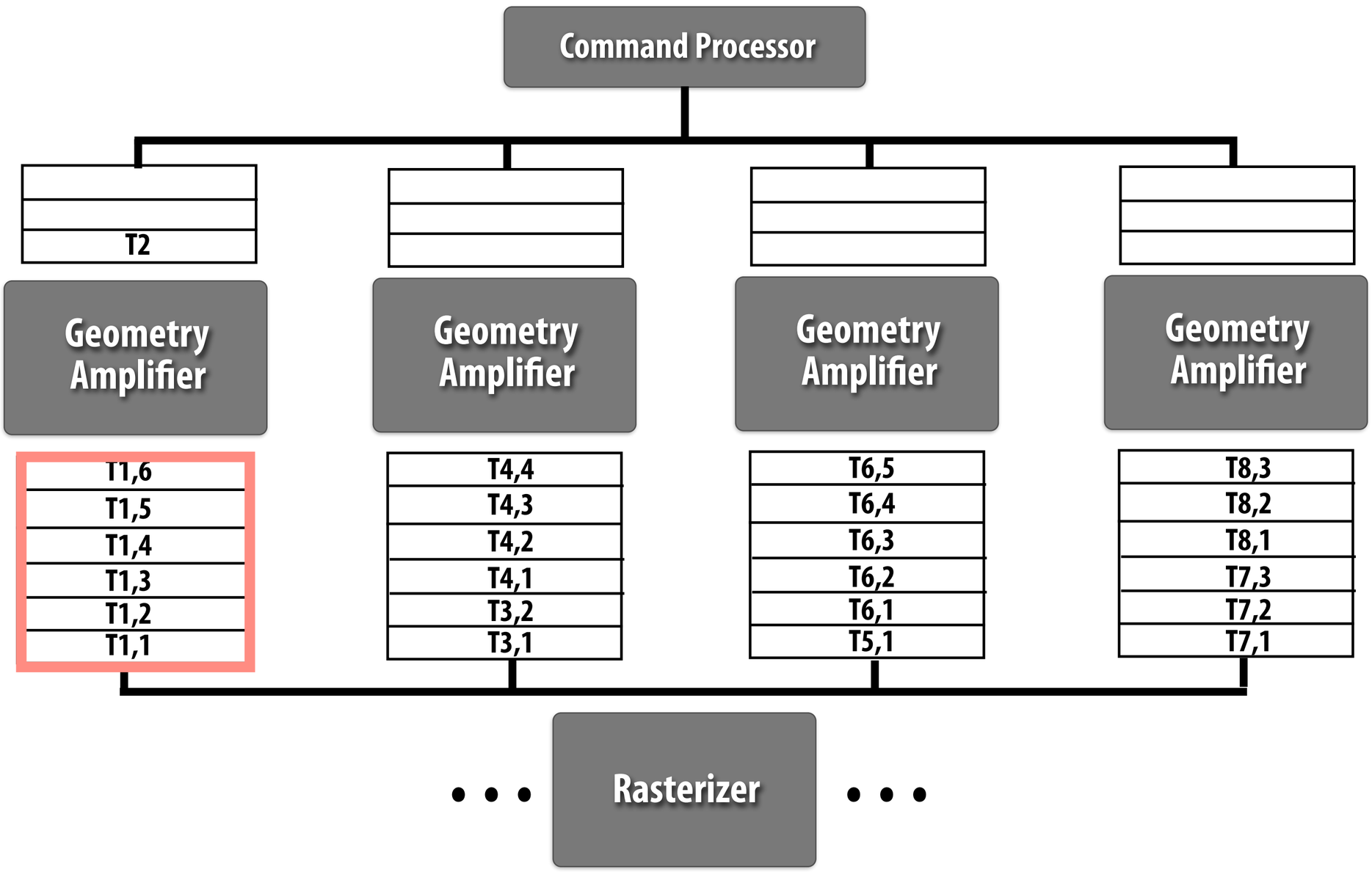
Consider case of large amplification when processing T1



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.**

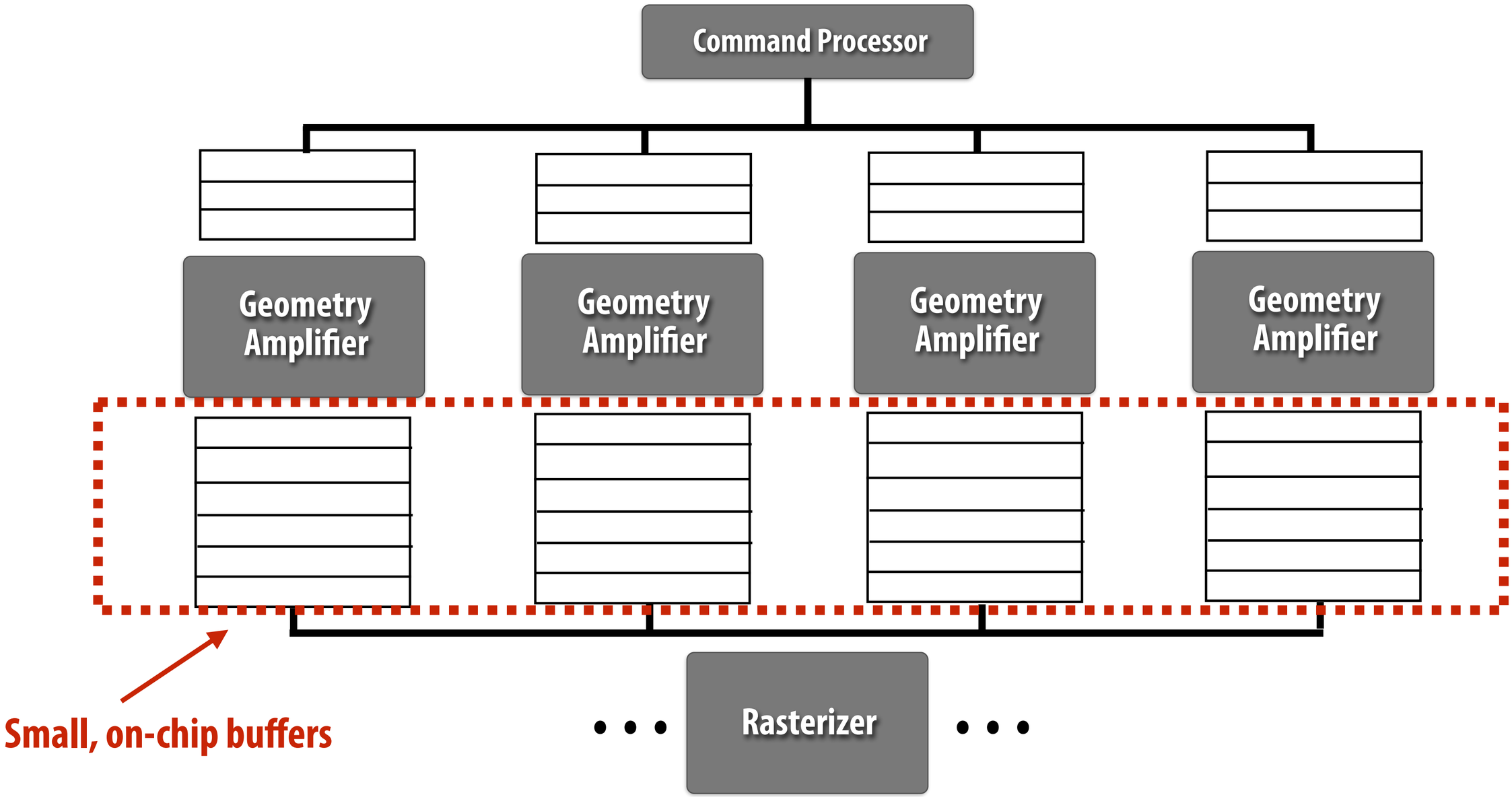
Thought experiment: design a scheduling strategy for this case



1. Design a solution that is high-performance when the expected amount of data amplification is low?
2. Design a solution that is high-performance 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.

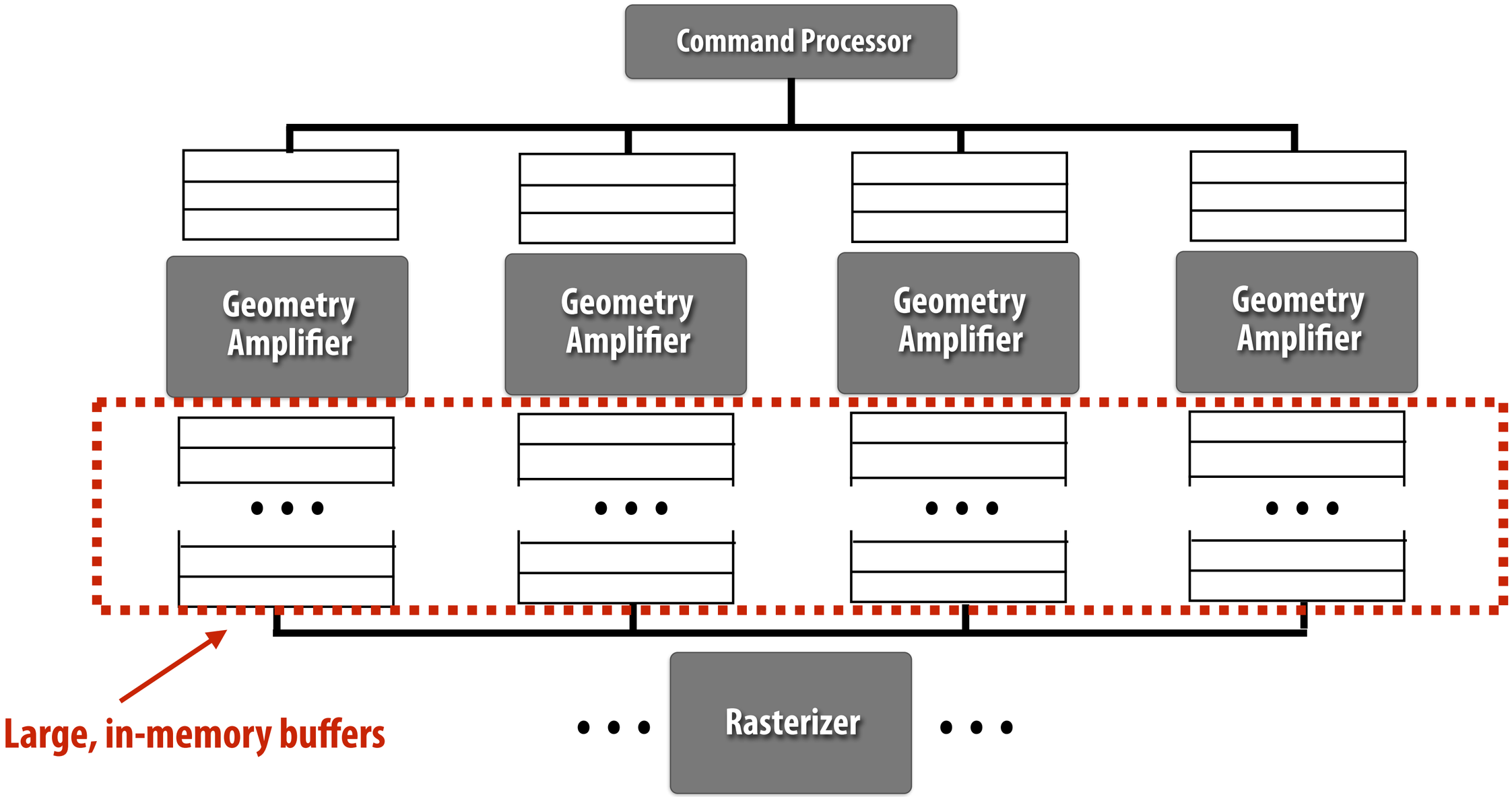
Implementation 1: fixed on-chip storage



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.**

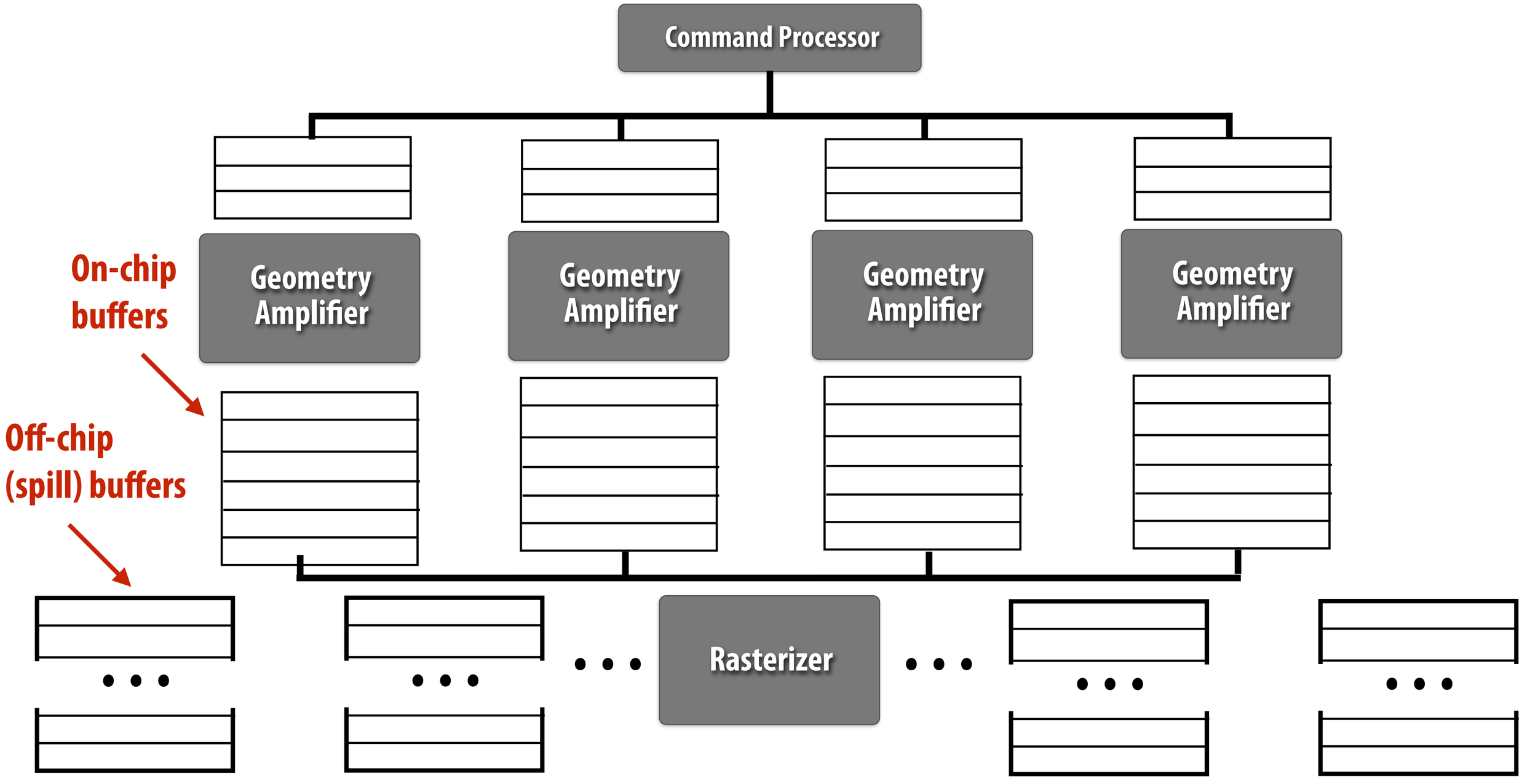
Implementation 2: worst-case allocation



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?**

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)