GPU architecture II: Scheduling the graphics pipeline

Mike Houston, AMD / Stanford
Aaron Lefohn, Intel / University of Washington

Notes

- The front half of this talk is almost verbatim from:

  “*Keeping Many Cores Busy: Scheduling the Graphics Pipeline*” by Jonathan Ragan-Kelley from SIGGRAPH Beyond Programmable Shading, 2010
- I’ll be adding my own twists from my background on working on some of the hardware in the talk

This talk

- How to think about scheduling GPU-style pipelines
  Four constraints which drive scheduling decisions

- Examples of these concepts in real GPU designs

- Goals
  - Know why GPUs, APIs impose the *constraints* they do.
  - Develop intuition for *what they can do well*.
  - Understand key patterns for *building your own pipelines*.

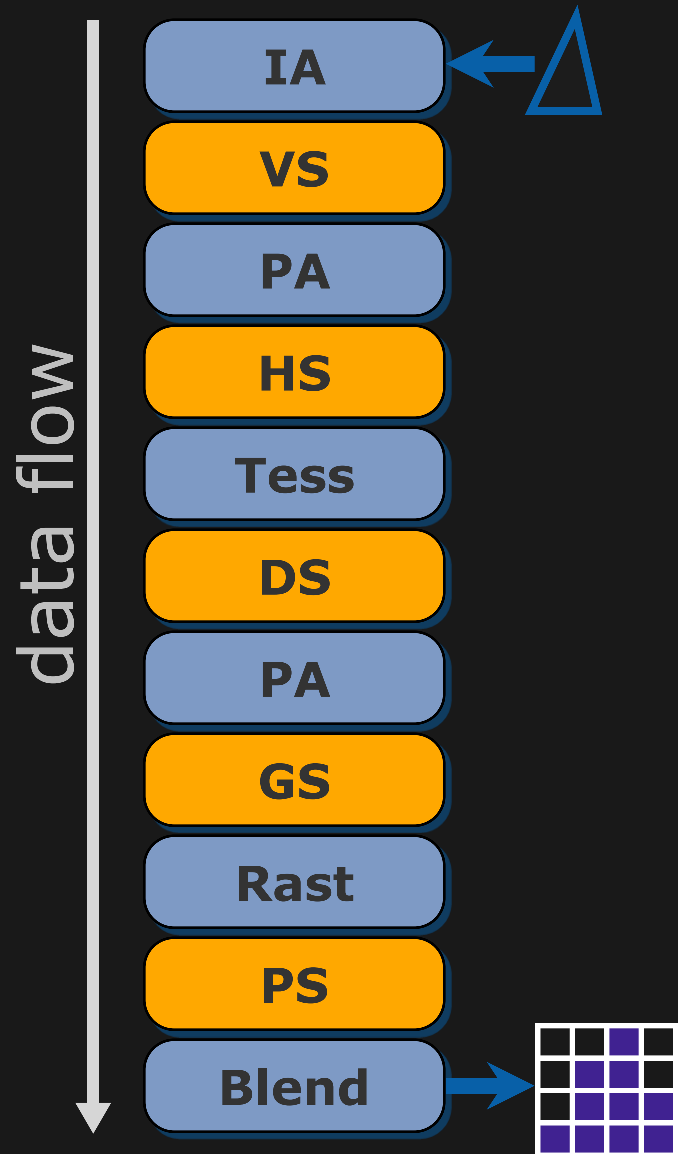
- Dig into ATI Radeon™ HD 5800 Series (“Cypress”) architecture deep dive
  - What questions do you have?

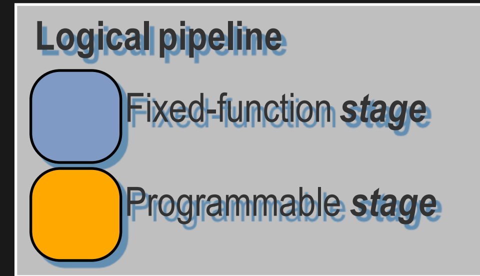
First, a definition

Scheduling [n.]:

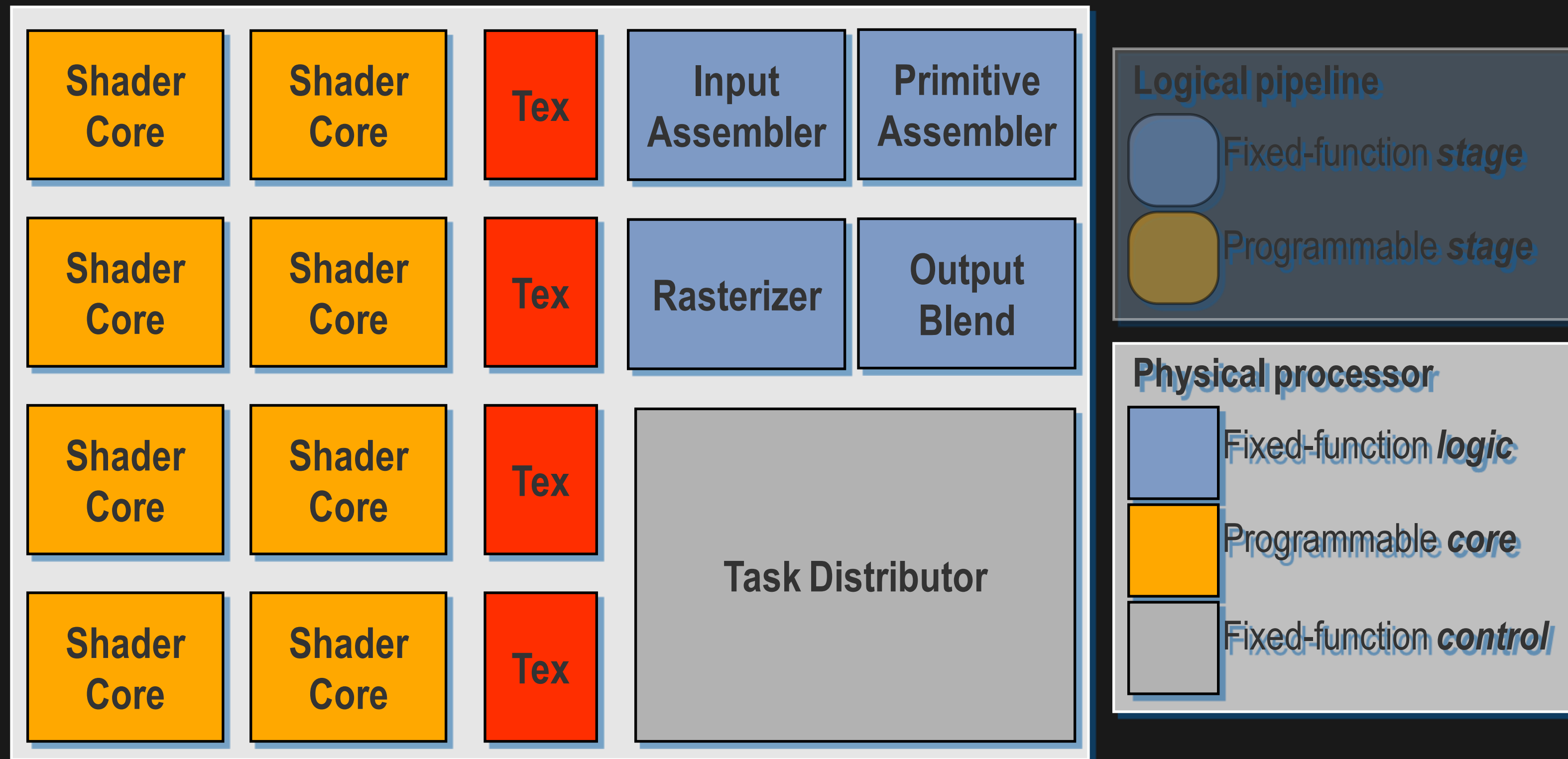
Assigning **computations** and **data** to resources in **space** and **time**.

The workload: Direct3D

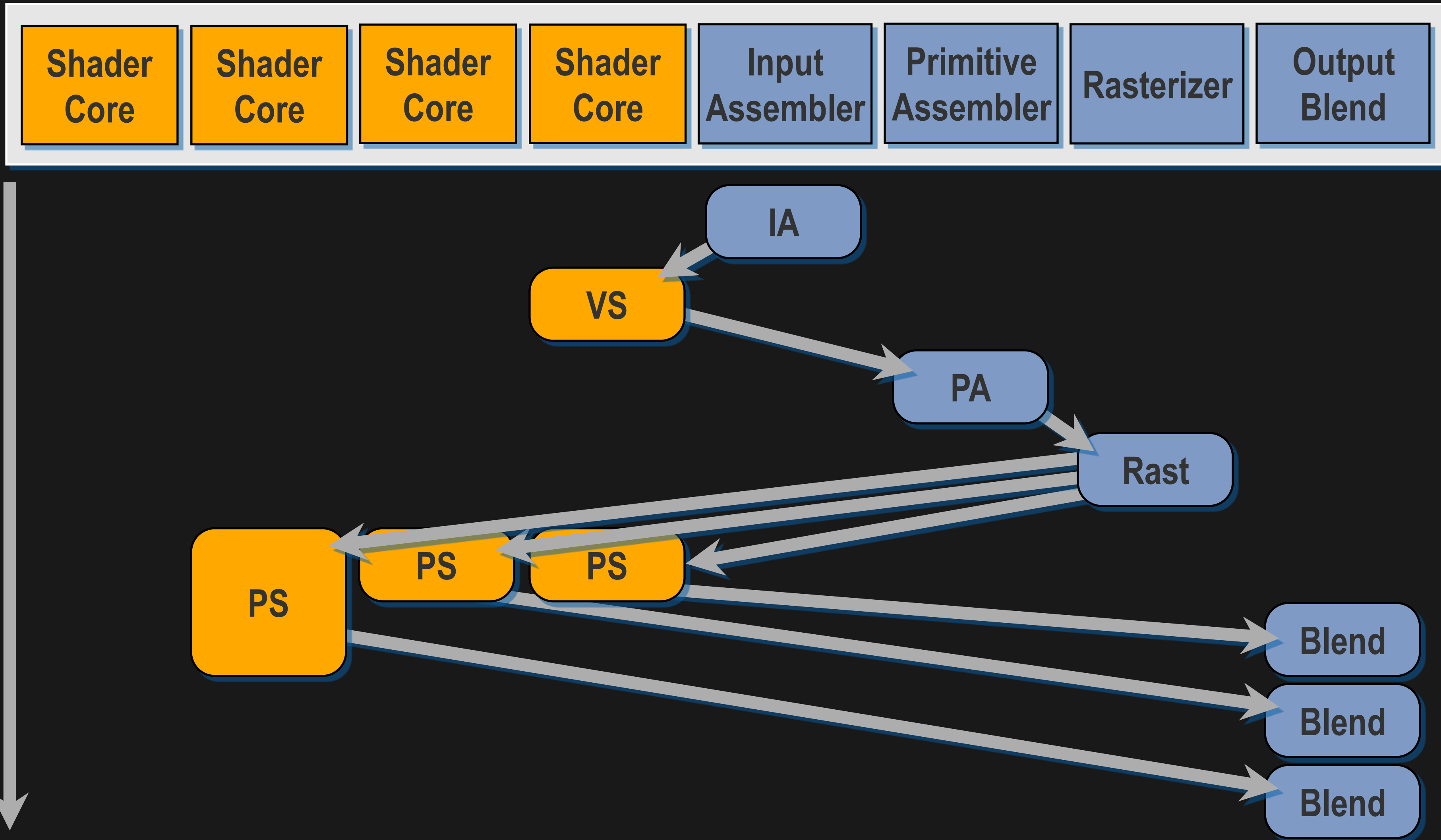




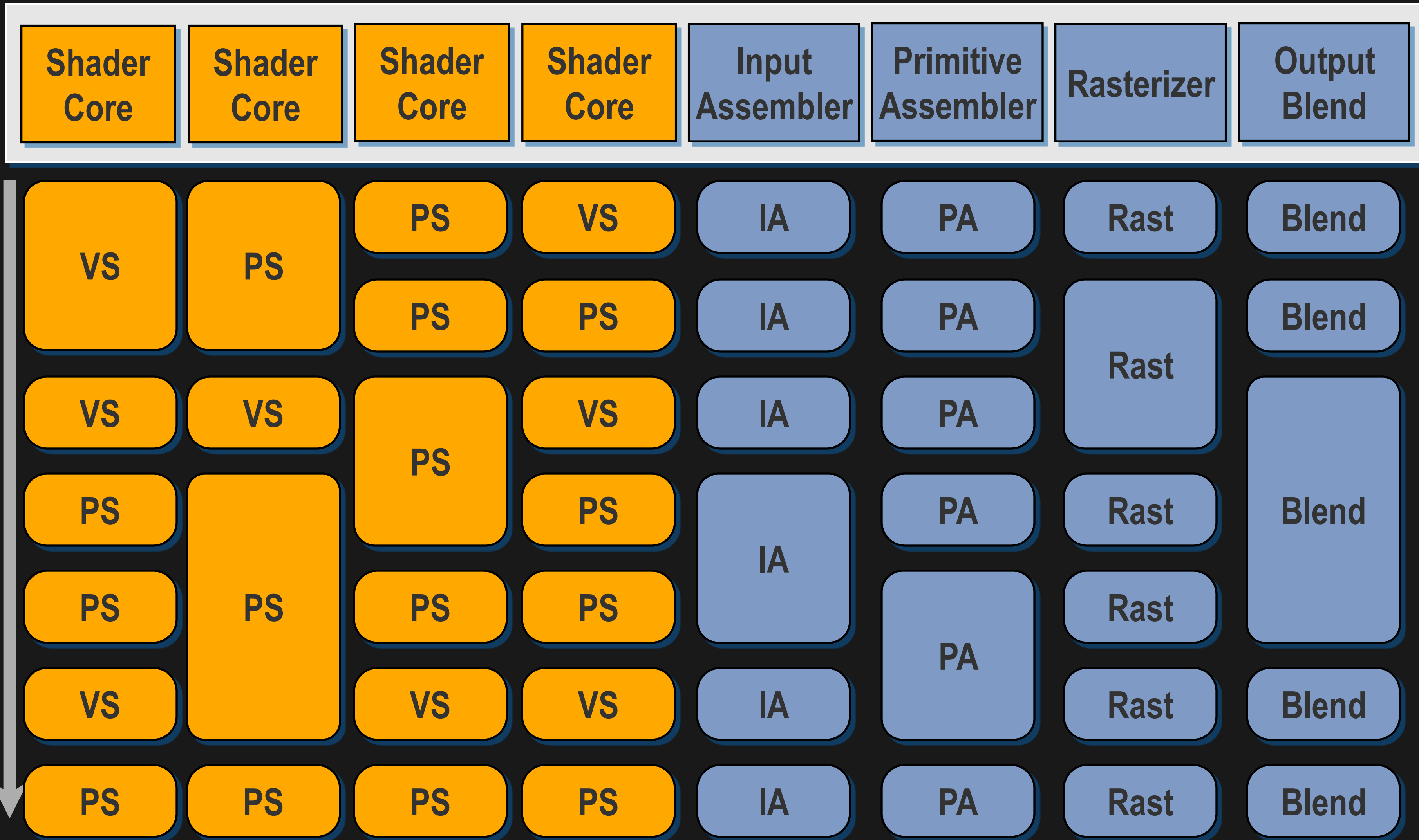
The machine: a modern GPU



Scheduling a draw call as a series of tasks



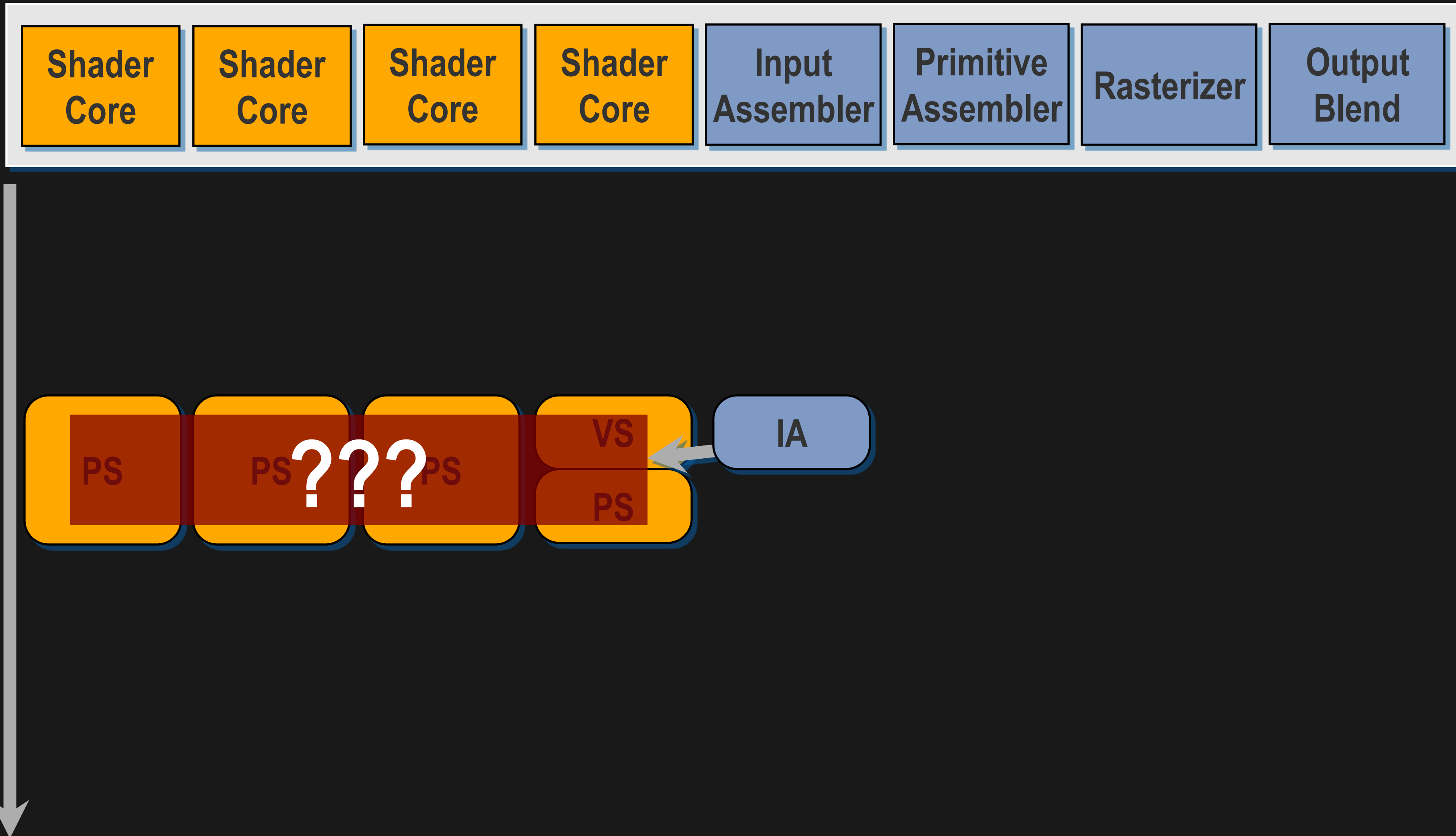
An efficient schedule keeps hardware busy



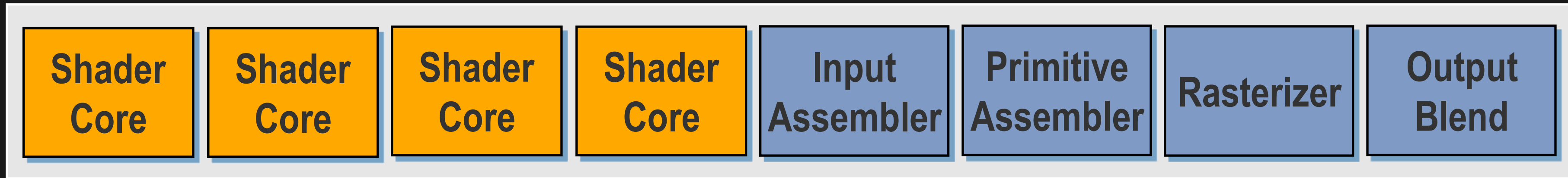
Choosing which tasks to run when (and where)

- Resource constraints
  - Tasks can only execute when there are sufficient resources for their computation *and* their data.
- Coherence
  - Control coherence is essential to shader core efficiency.
  - Data coherence is essential to memory and communication efficiency.
- Load balance
  - Irregularity in execution time create bubbles in the pipeline schedule.
- Ordering
  - Graphics APIs define strict ordering semantics, which restrict possible schedules.

Resource constraints limit scheduling options



Resource constraints limit scheduling options



***Key concept*:** Pre-allocation of resources helps guarantee forward progress.

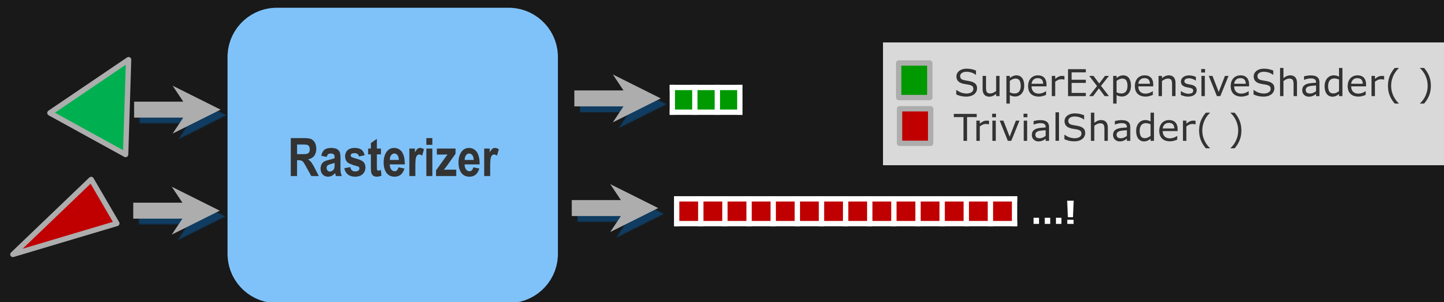
Coherence is a balancing act

Intrinsic tension between:

Horizontal (control, fetch) coherence and
Vertical (producer-consumer) locality.

Locality and **Load Balance**.

Graphics workloads are irregular



But: Shaders are optimized for regular, self-similar work.

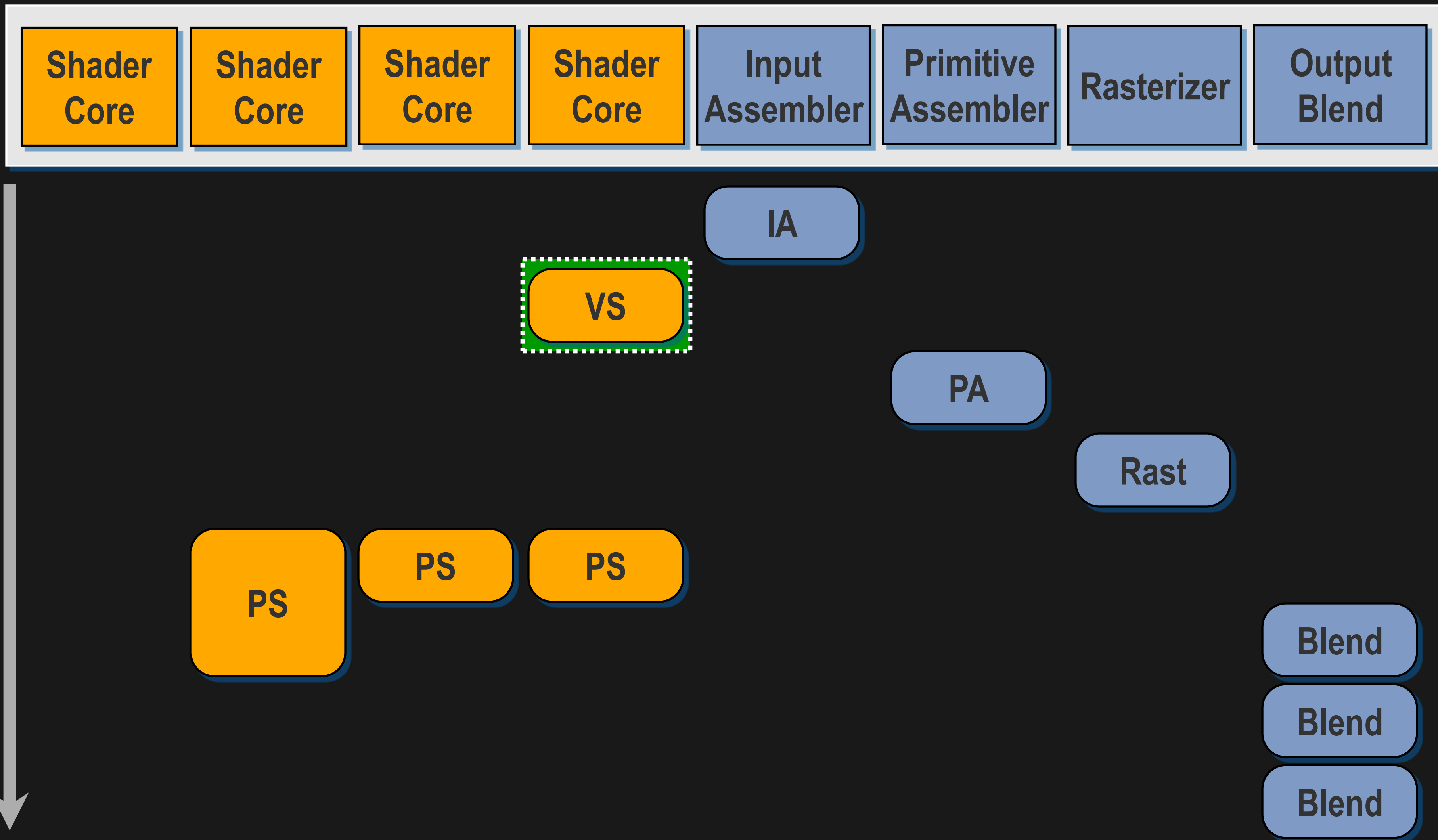
Imbalanced work creates *bubbles in the task schedule.*

Solution:

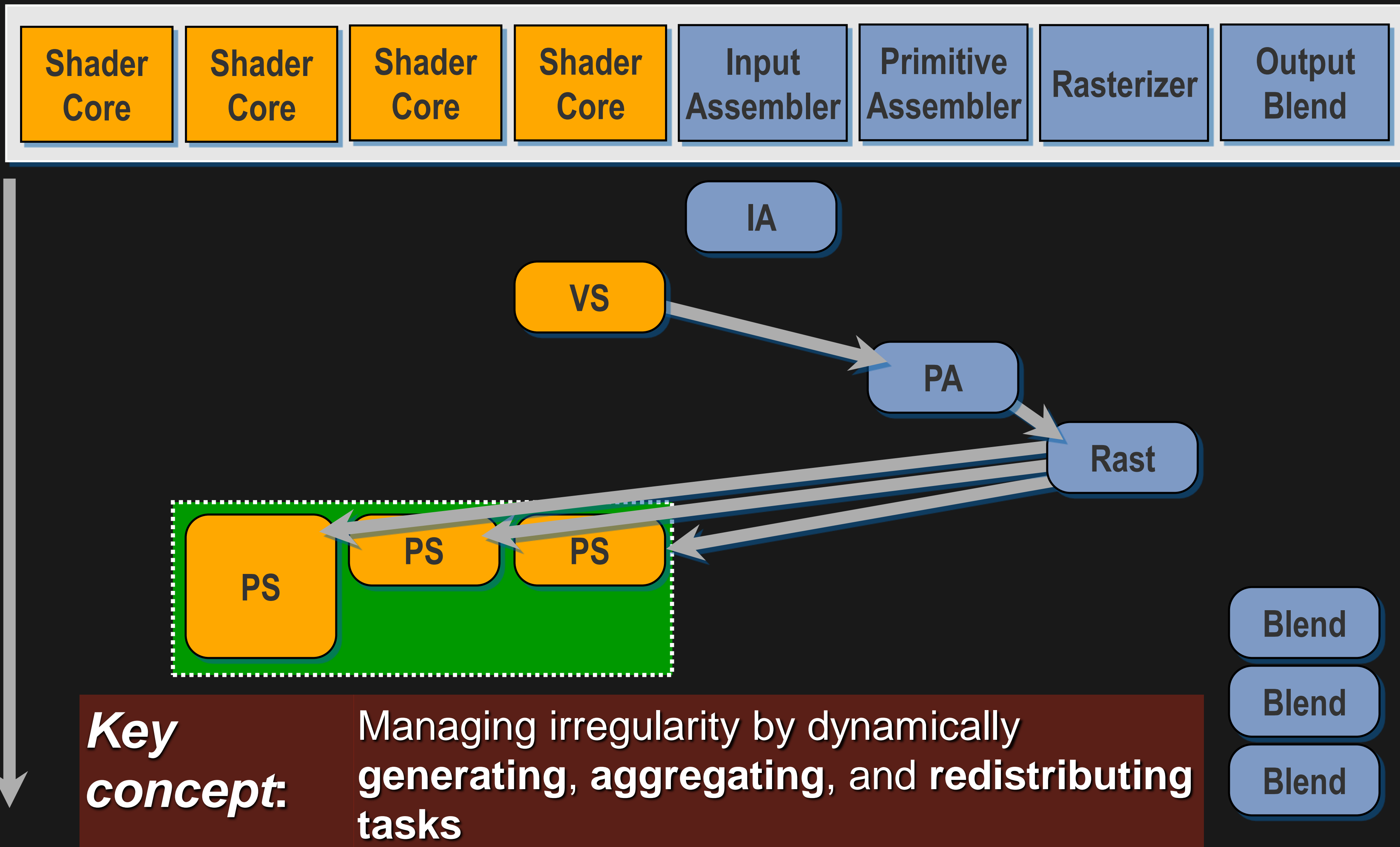
Dynamically **generating and aggregating tasks** isolates irregularity and recaptures **coherence.**

Redistributing tasks restores load balance.

Redistribution after irregular amplification



Redistribution after irregular amplification



Ordering

Rule:

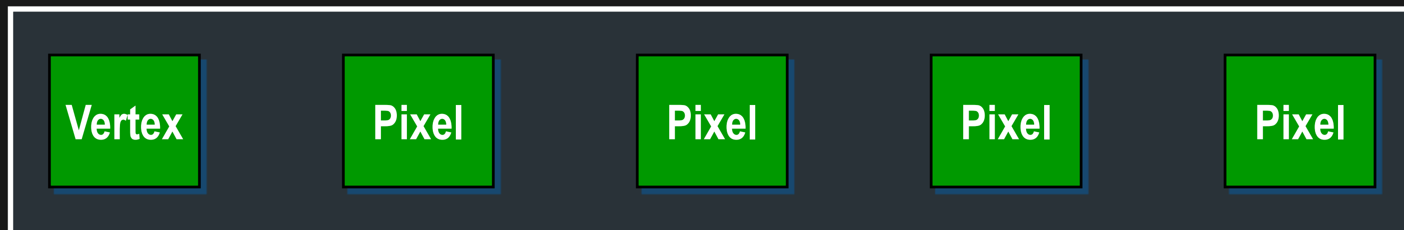
All framebuffer updates must appear as though all triangles were drawn in strict sequential order

Key concept:	Carefully structuring **task redistribution** to **maintain API ordering**.

Building a real pipeline

Static tile scheduling

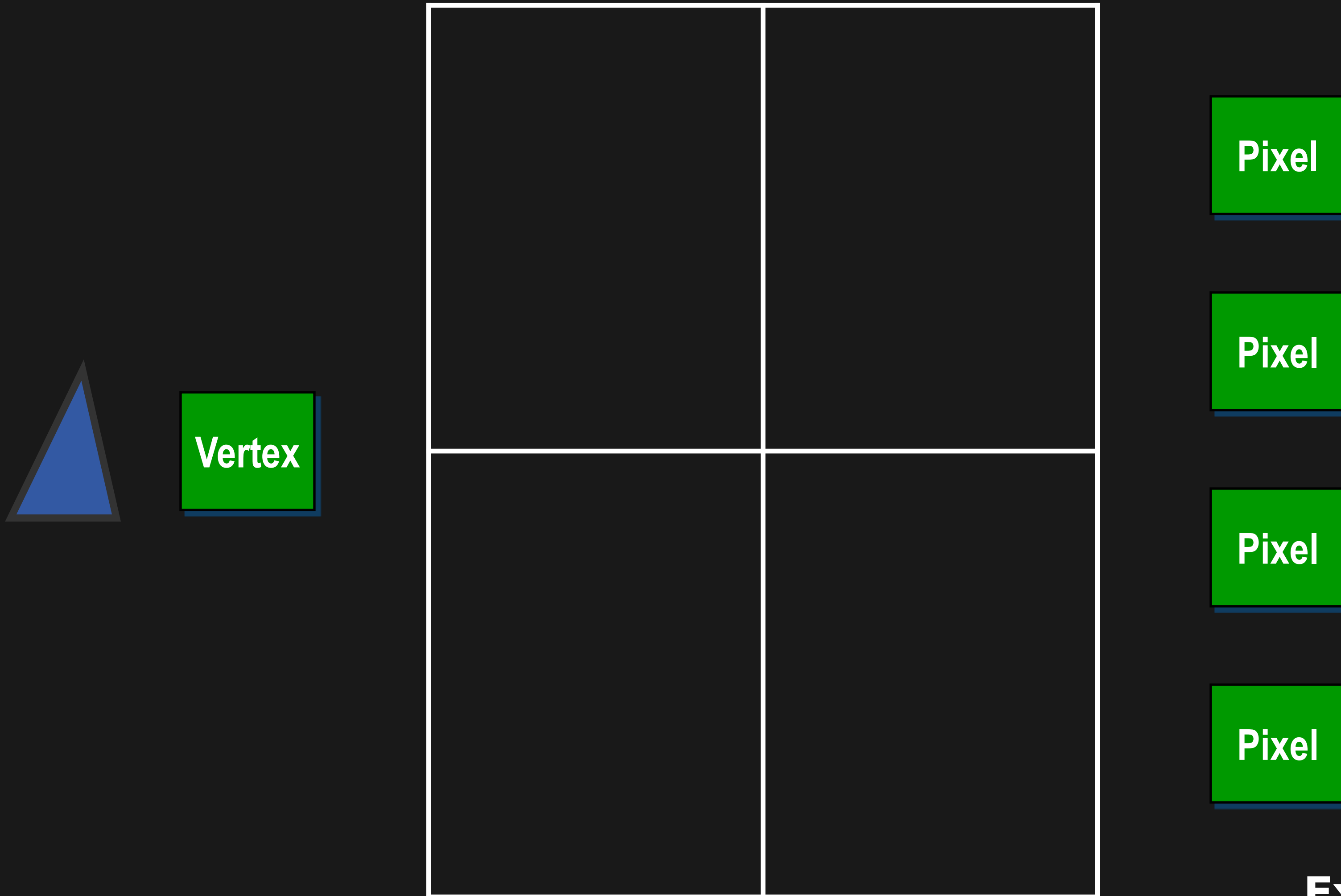
The simplest thing that could possibly work.



Multiple cores:
1 front-end
n back-end

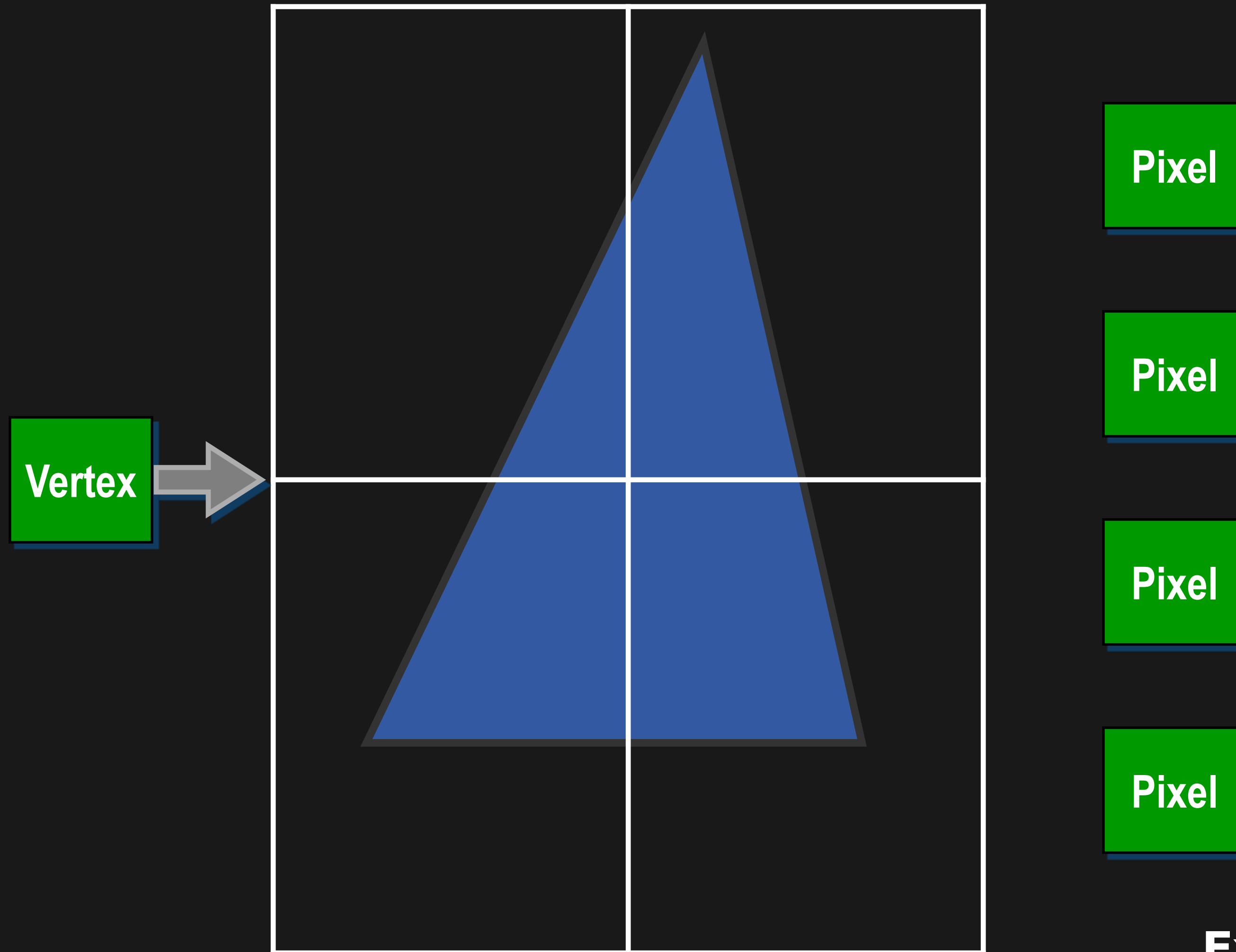
Exemplar:
ARM Mali 400

Static tile scheduling



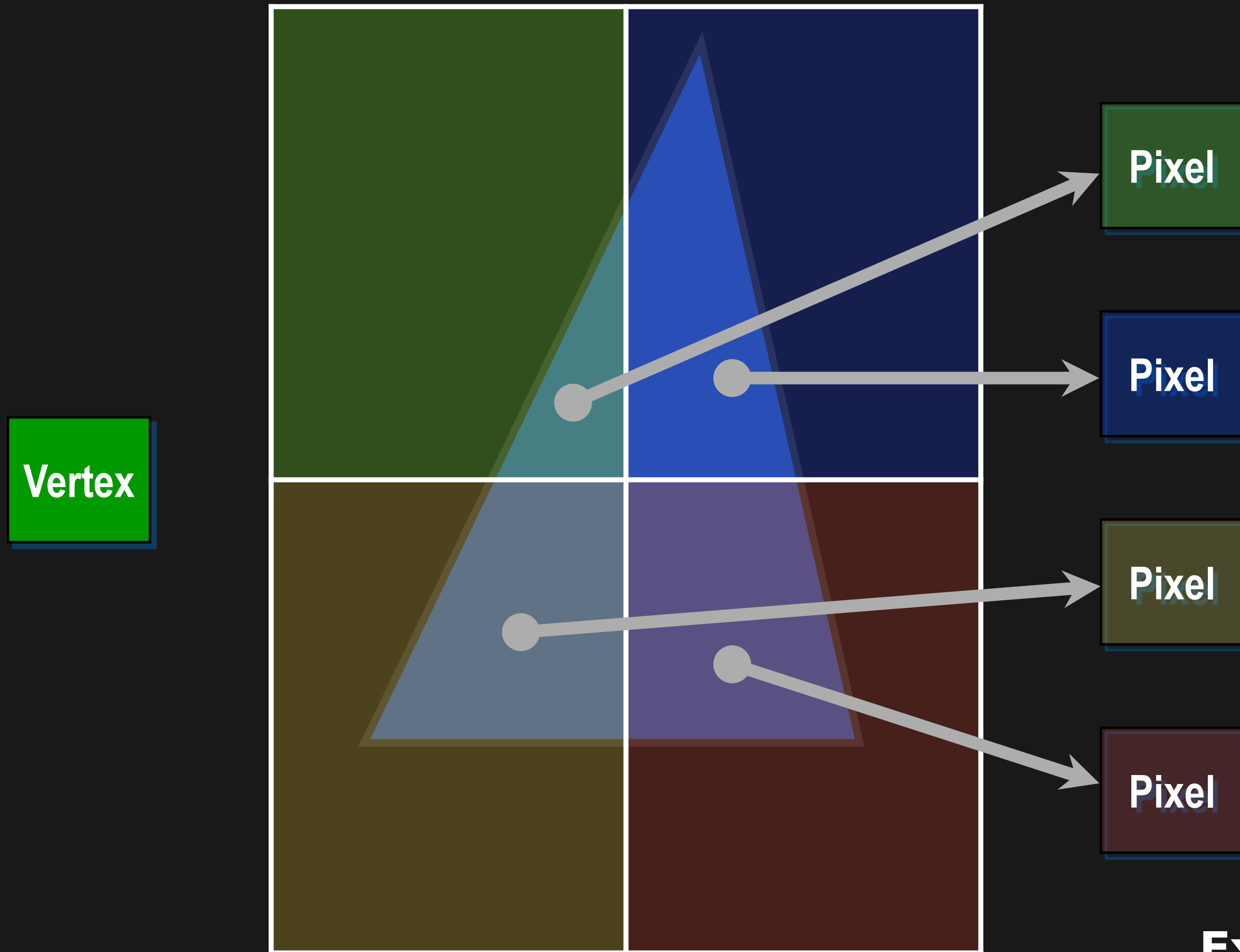
Exemplar:
ARM Mali 400

Static tile scheduling



Exemplar:
ARM Mali 400

Static tile scheduling



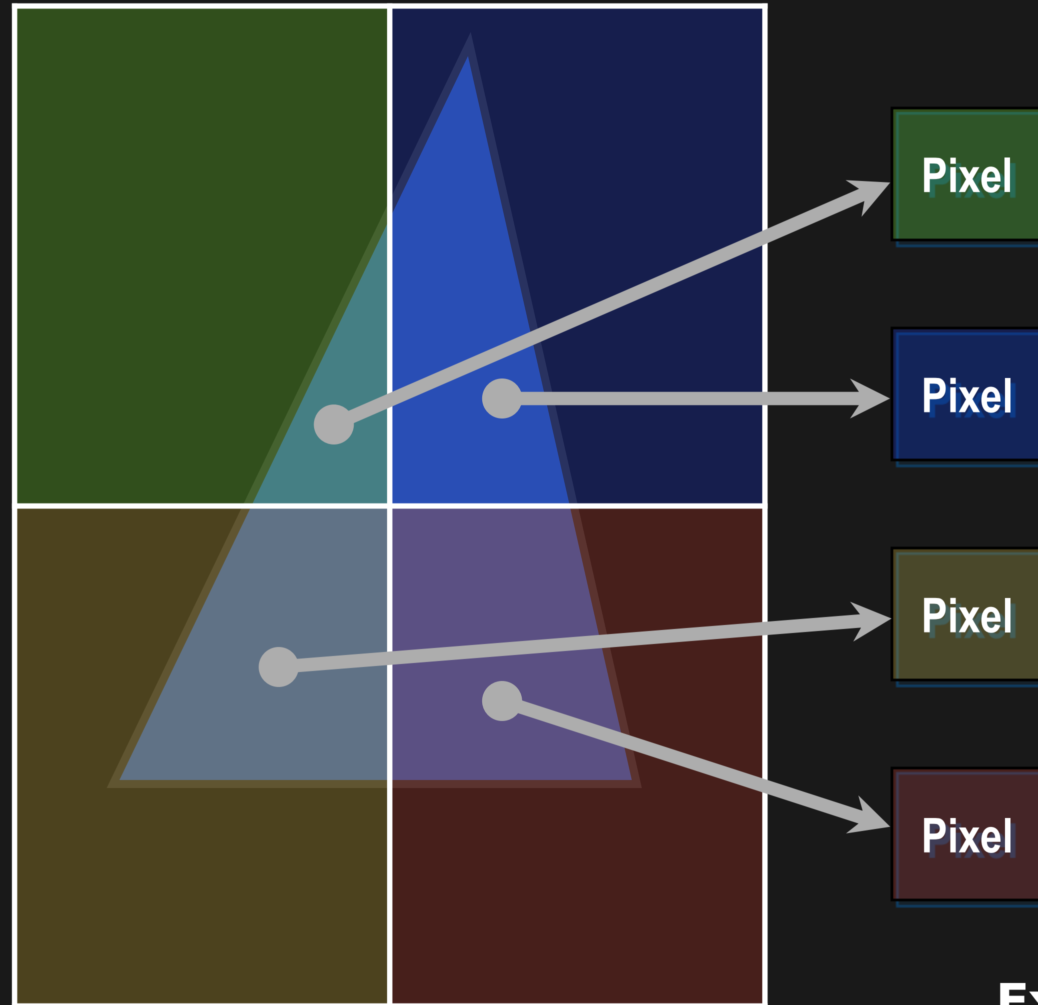
Exemplar:
ARM Mali 400

Static tile scheduling

Locality
captured within tiles

Resource constraints
static = simple

Ordering
single front-end, sequential processing within each tile



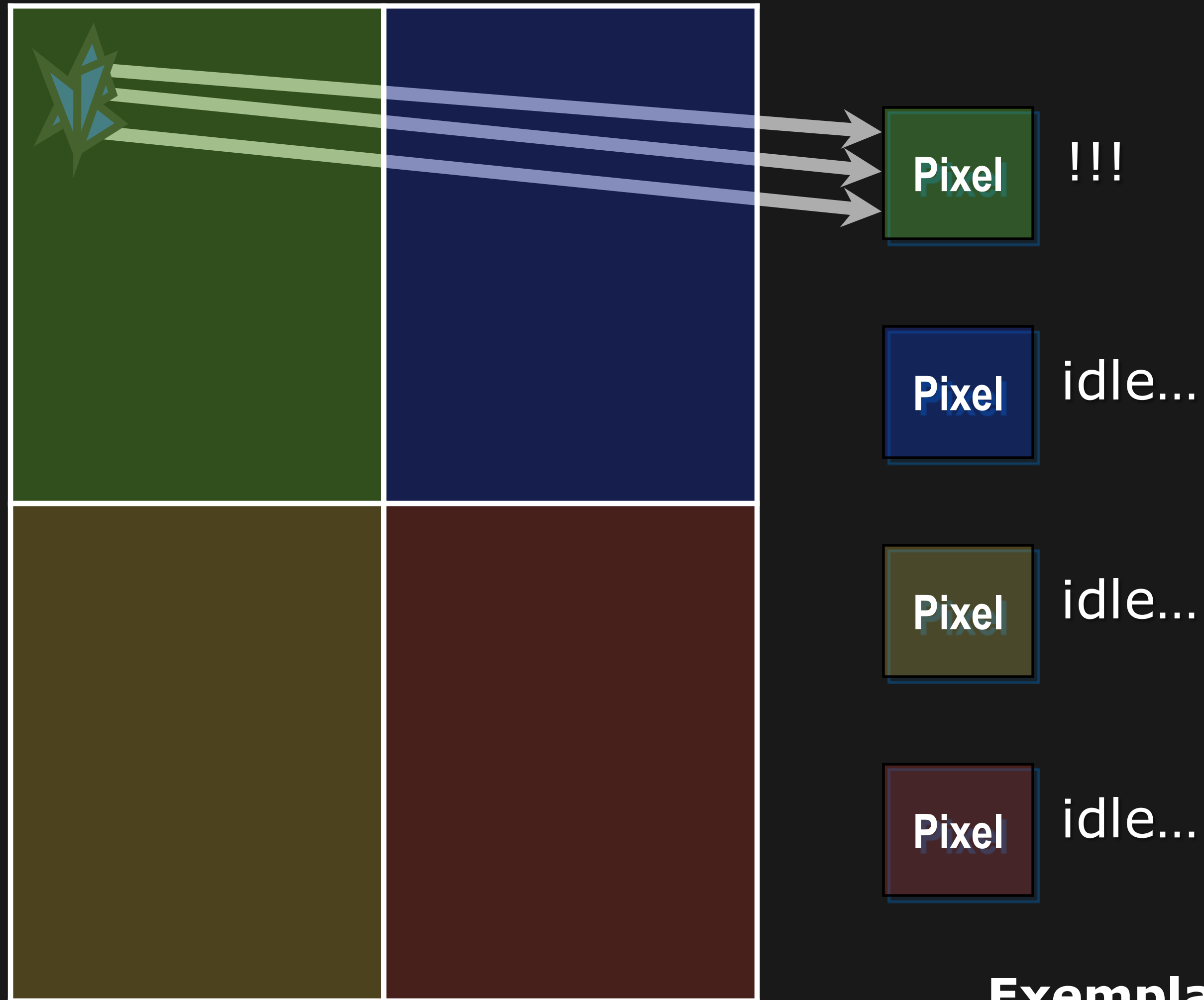
Exemplar:
ARM Mali 400

Static tile scheduling

The problem: load imbalance

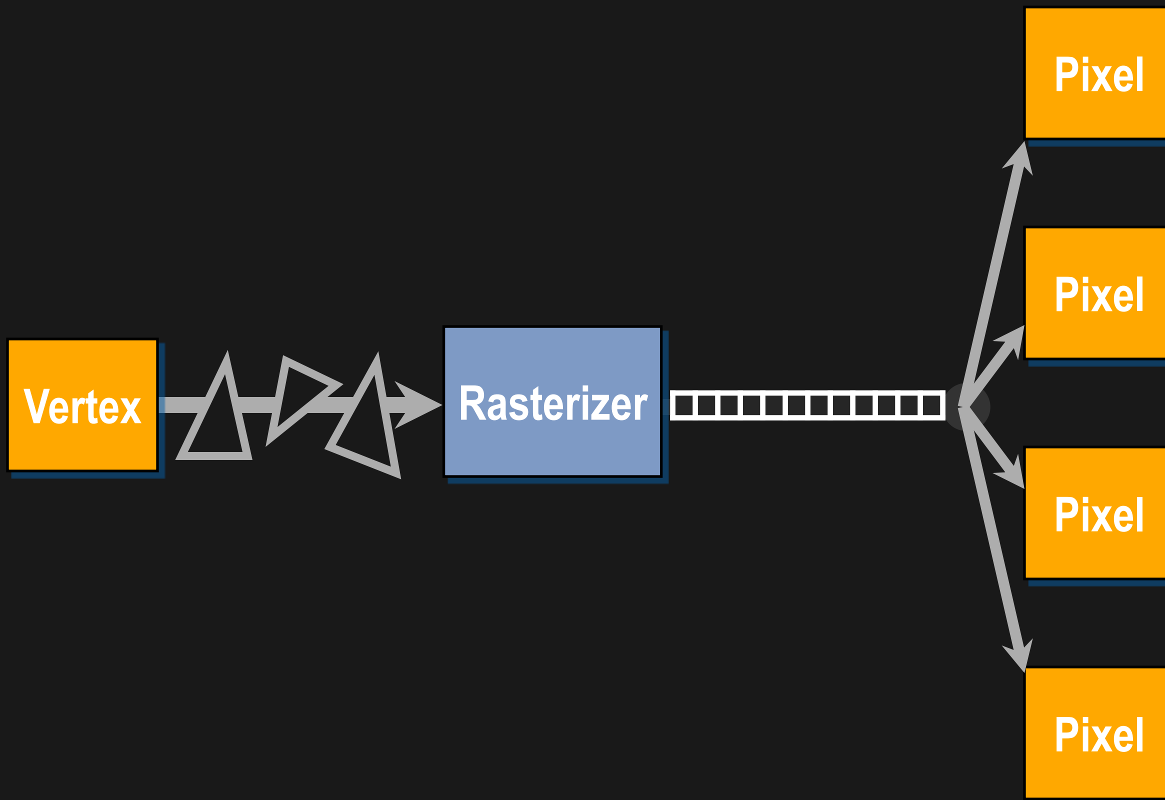
only one *task creation* point.

no *dynamic task redistribution.*



Exemplar:
ARM Mali 400

Sort-last fragment shading

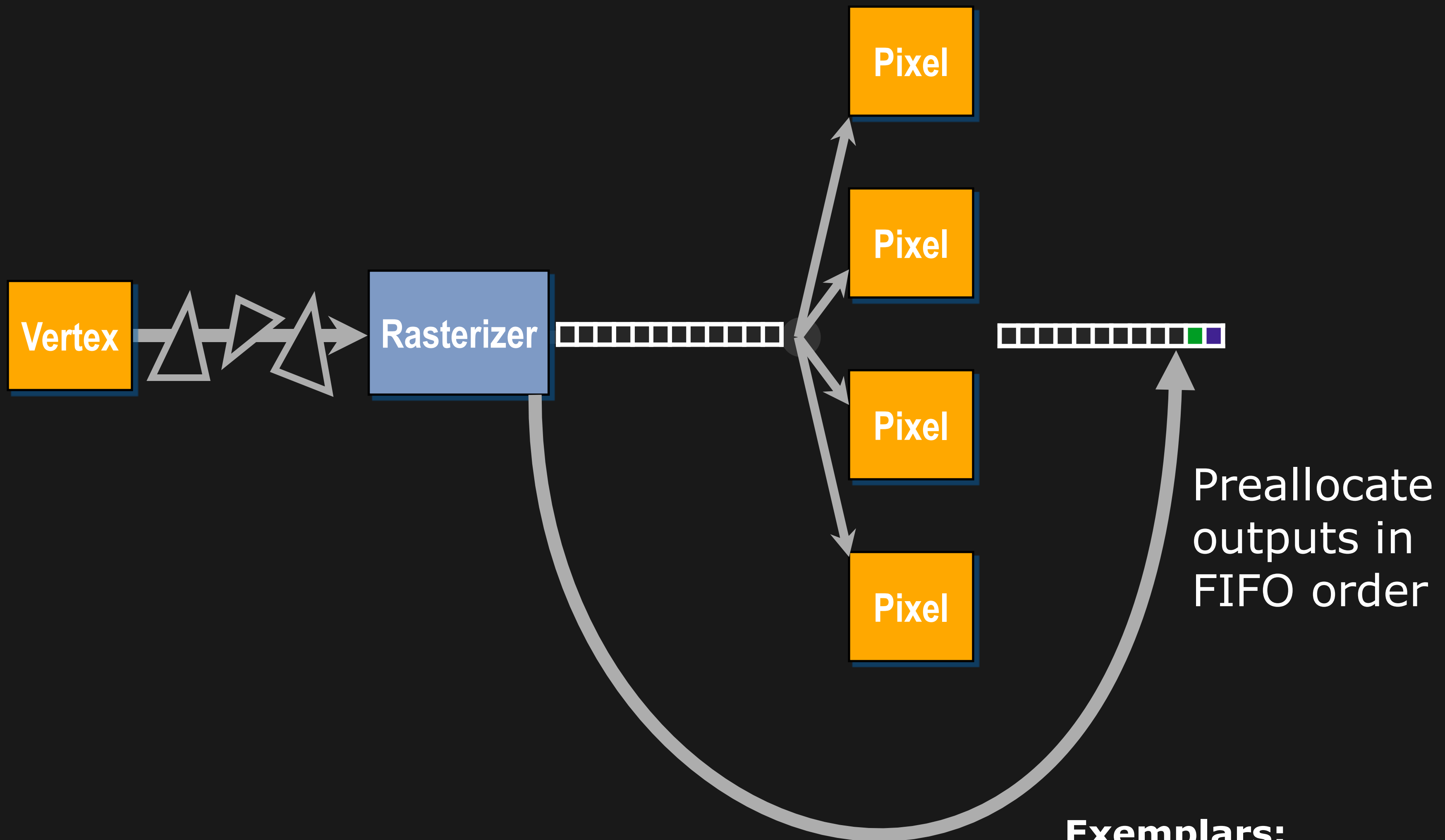


Redistribution restores fragment load balance.

But how can we maintain order?

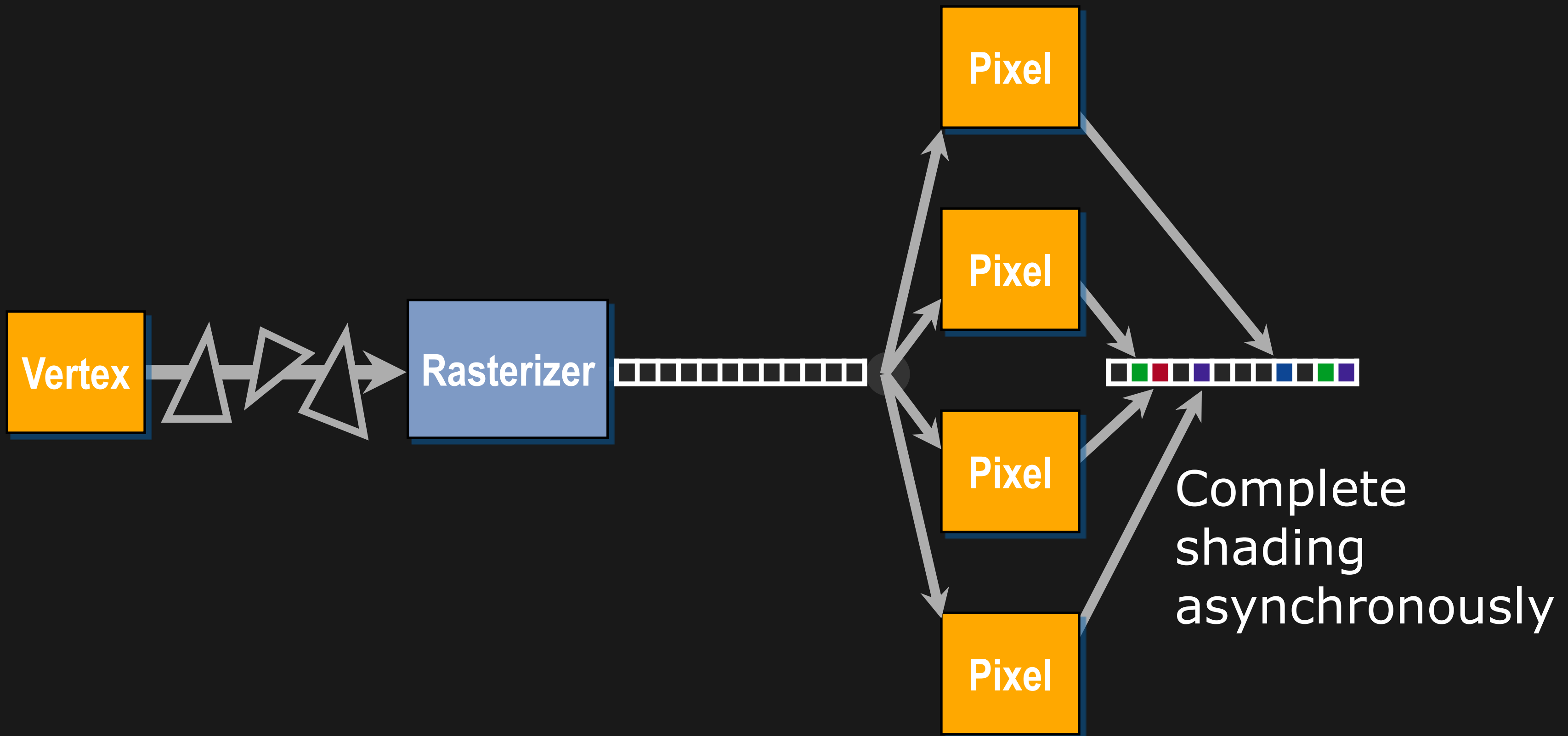
Exemplars:
NVIDIA G80, ATI RV770

Sort-last fragment shading



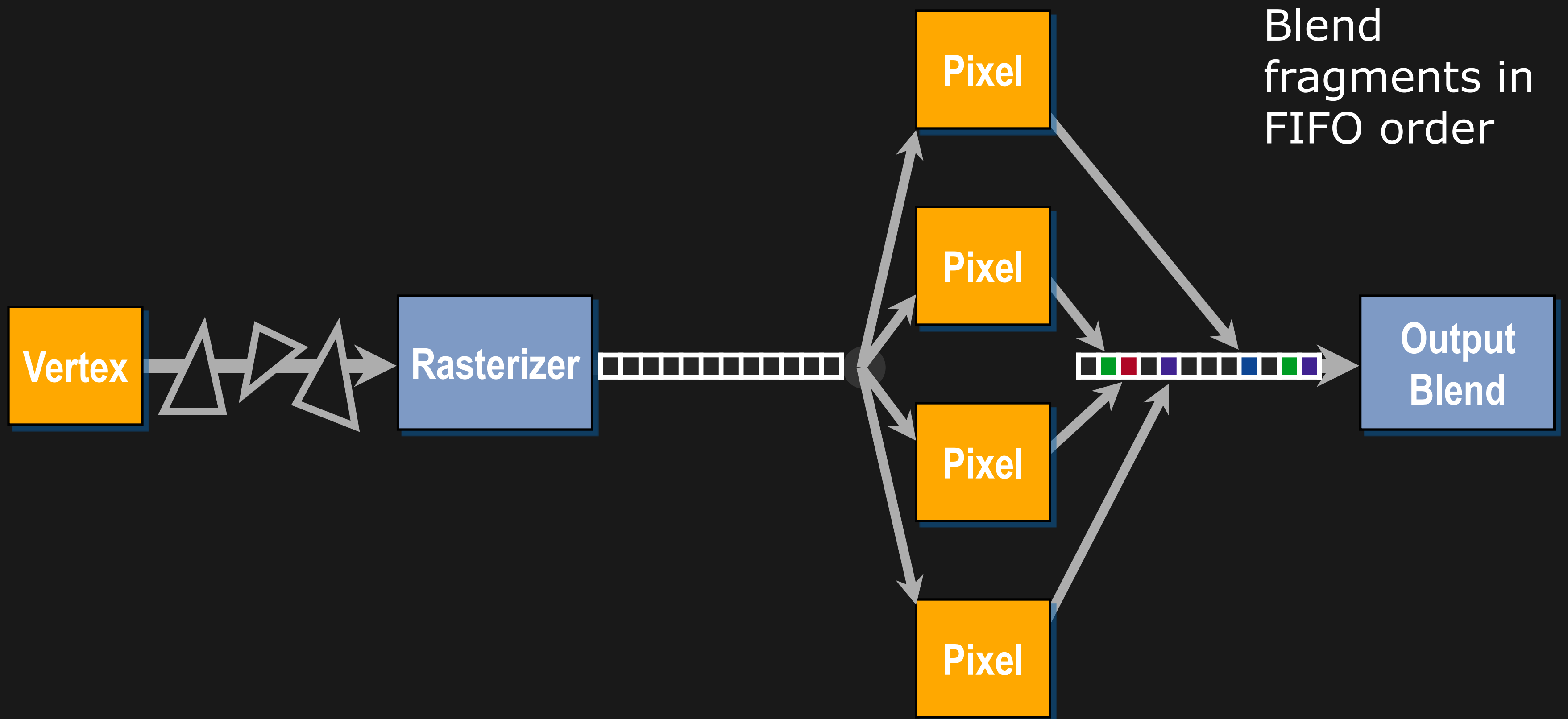
Exemplars:
NVIDIA G80, ATI RV770

Sort-last fragment shading



Exemplars:
NVIDIA G80, ATI RV770

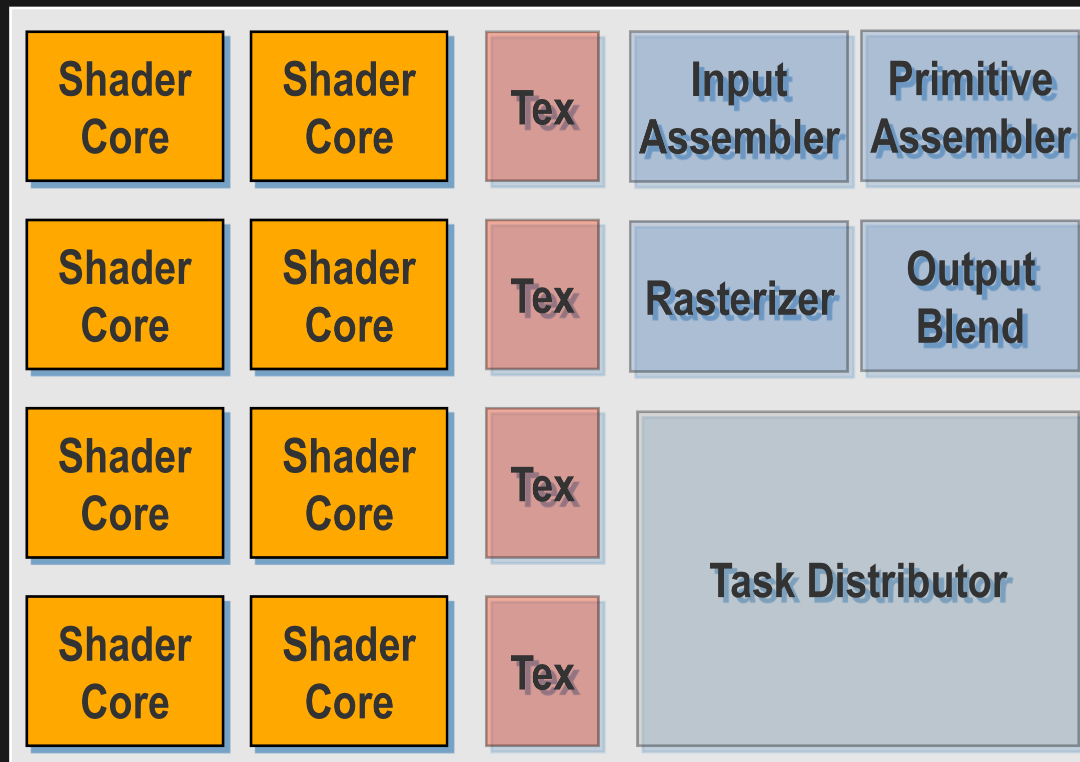
Sort-last fragment shading



Exemplars:
NVIDIA G80, ATI RV770

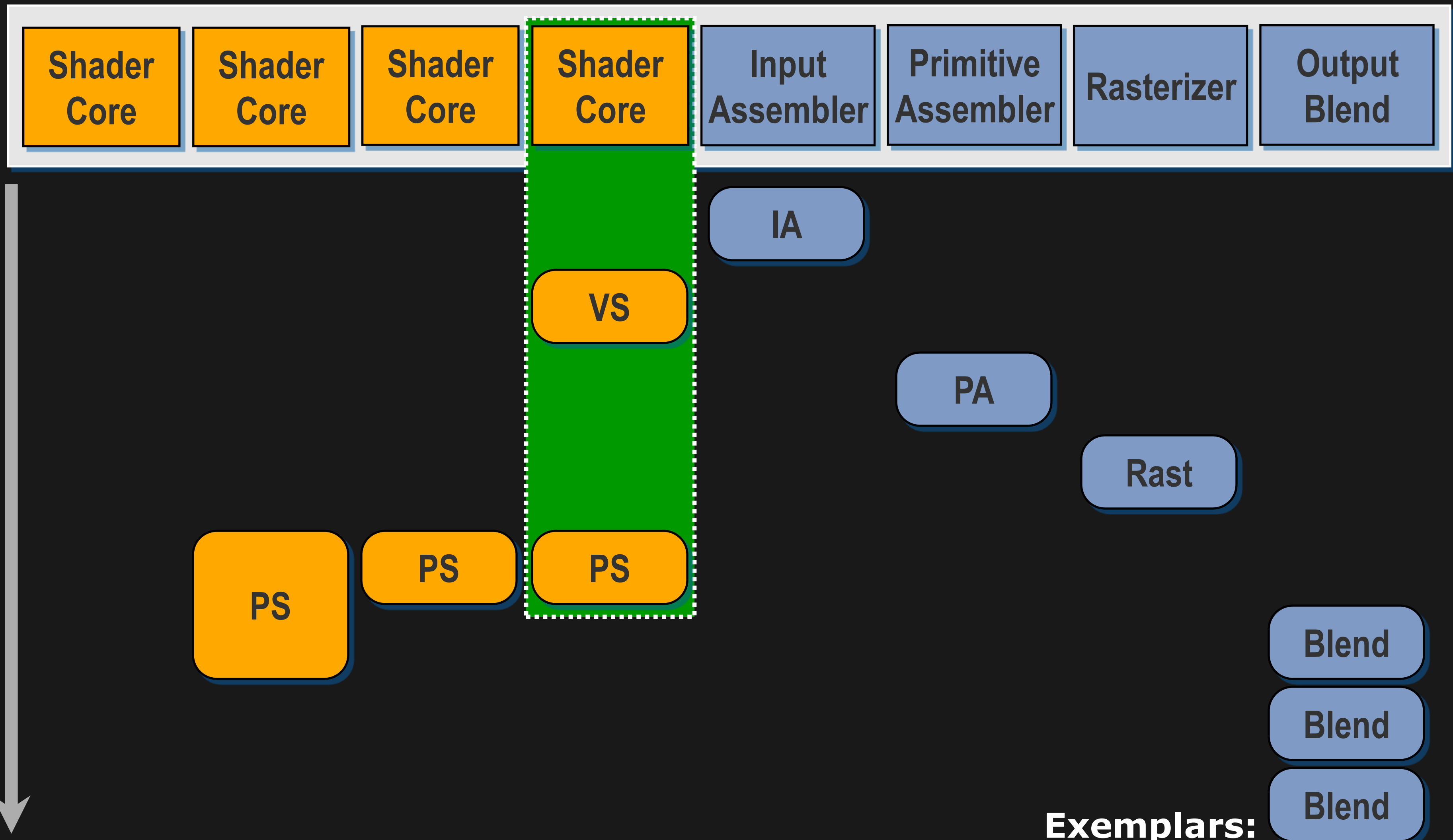
Unified shaders

Solve load balance by time-multiplexing different stages onto shared processors according to load



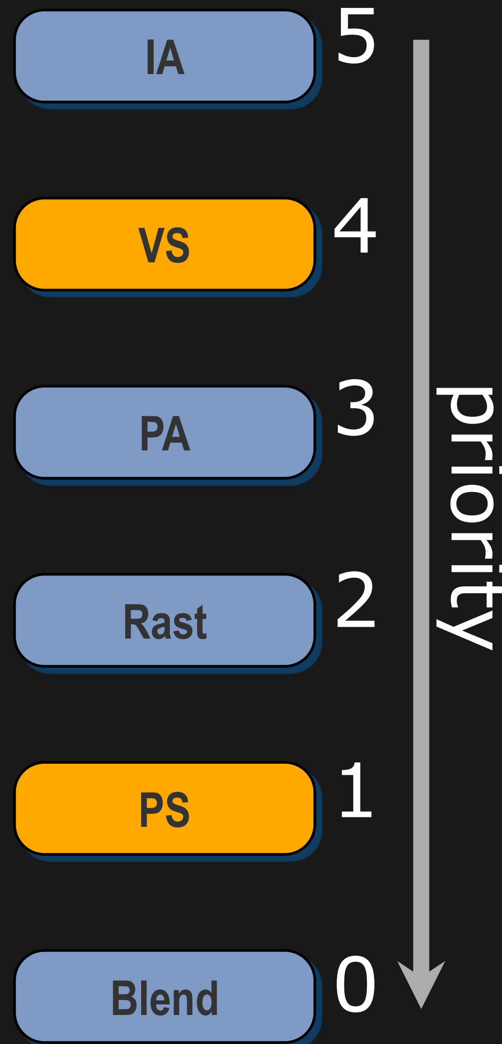
Exemplars:
NVIDIA G80, ATI RV770

Unified Shaders: time-multiplexing cores

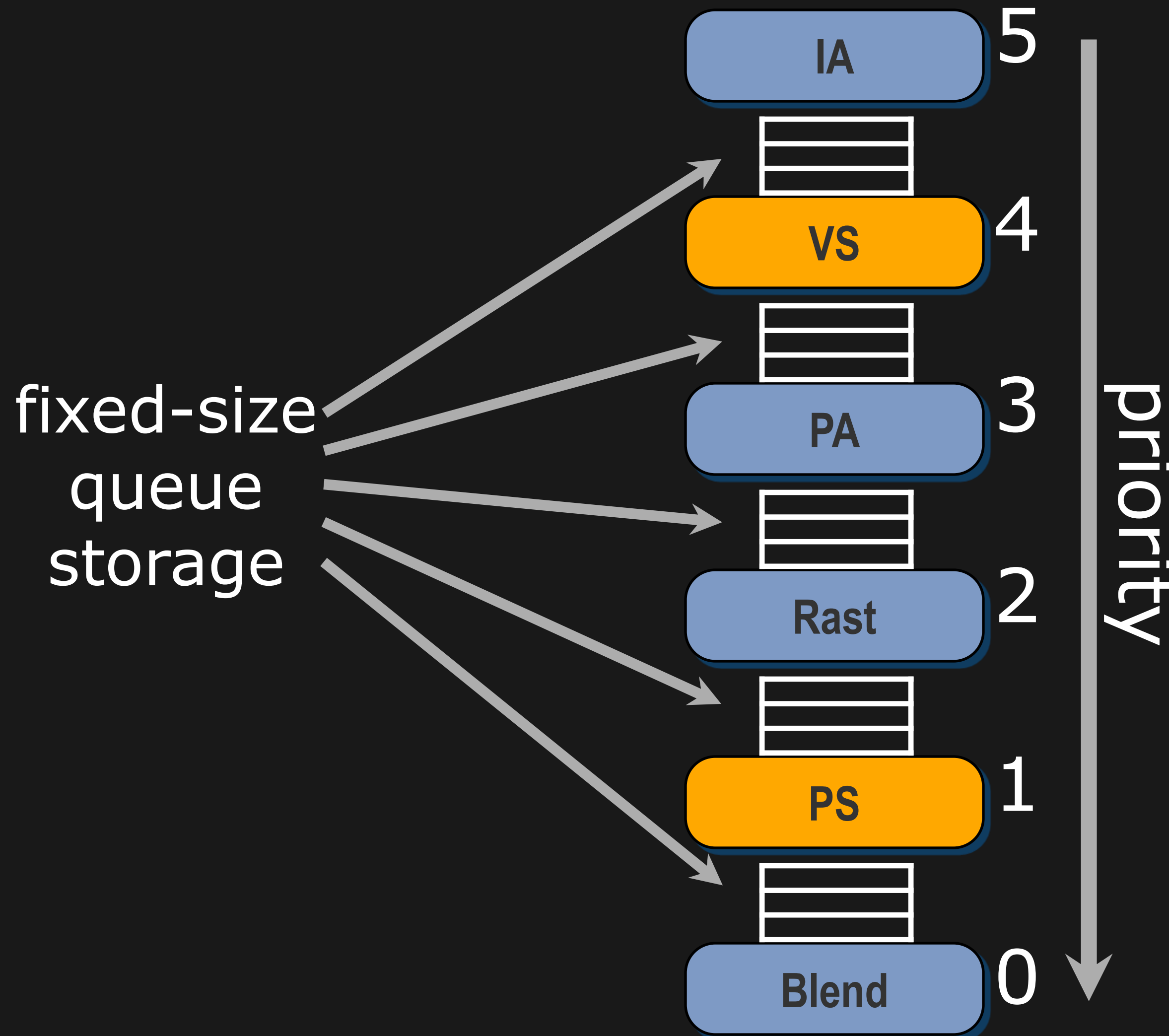


Exemplars:
NVIDIA G80, ATI RV770

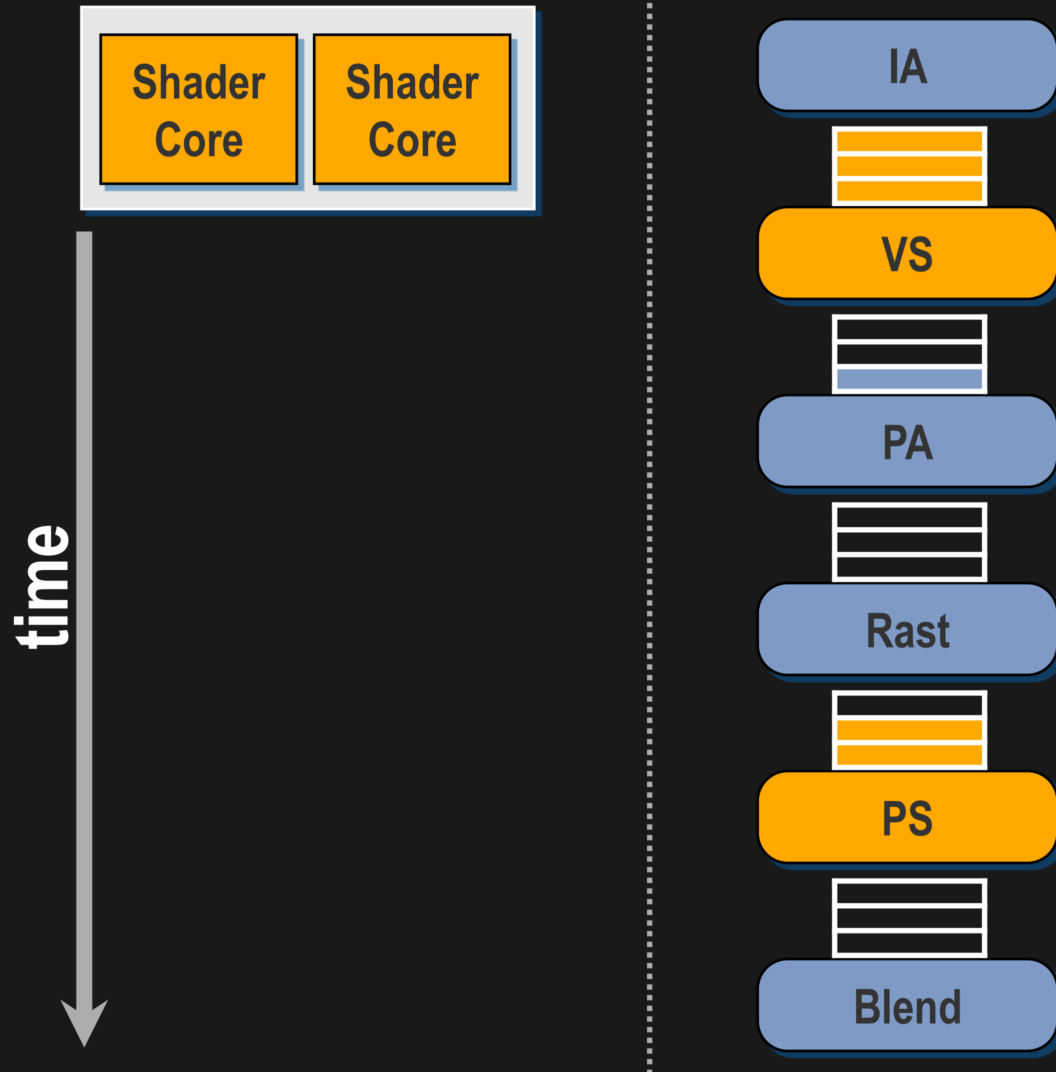
Prioritizing the logical pipeline



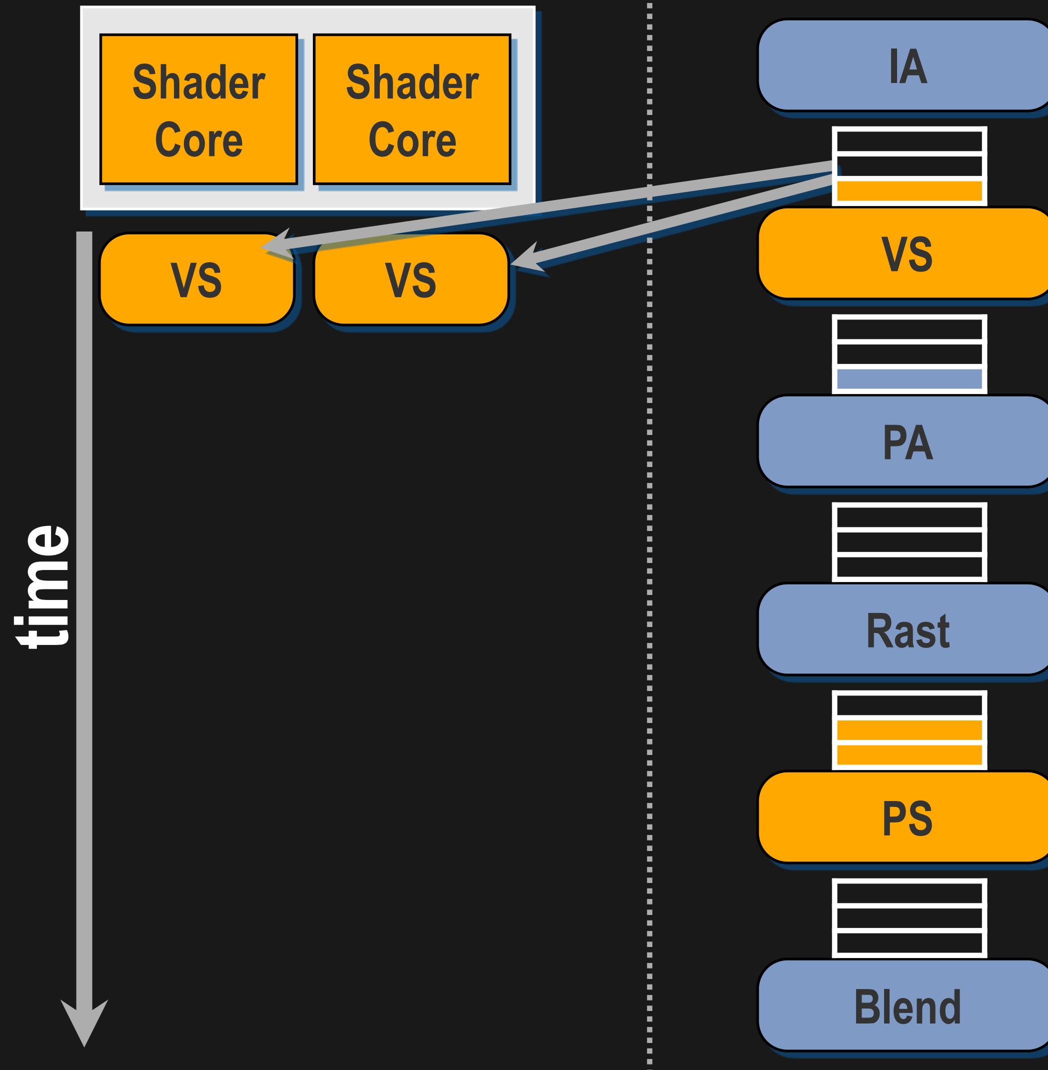
Prioritizing the logical pipeline



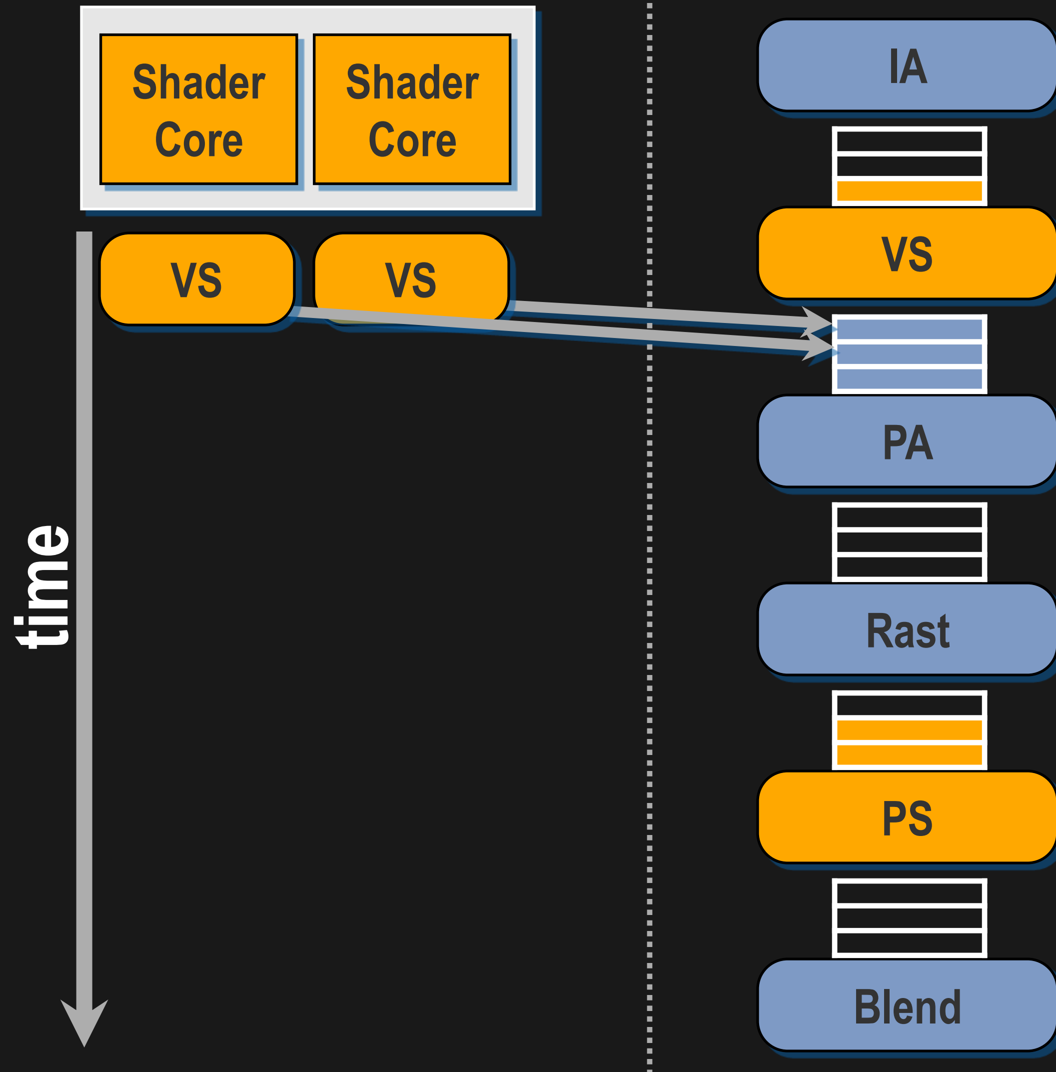
Scheduling the pipeline



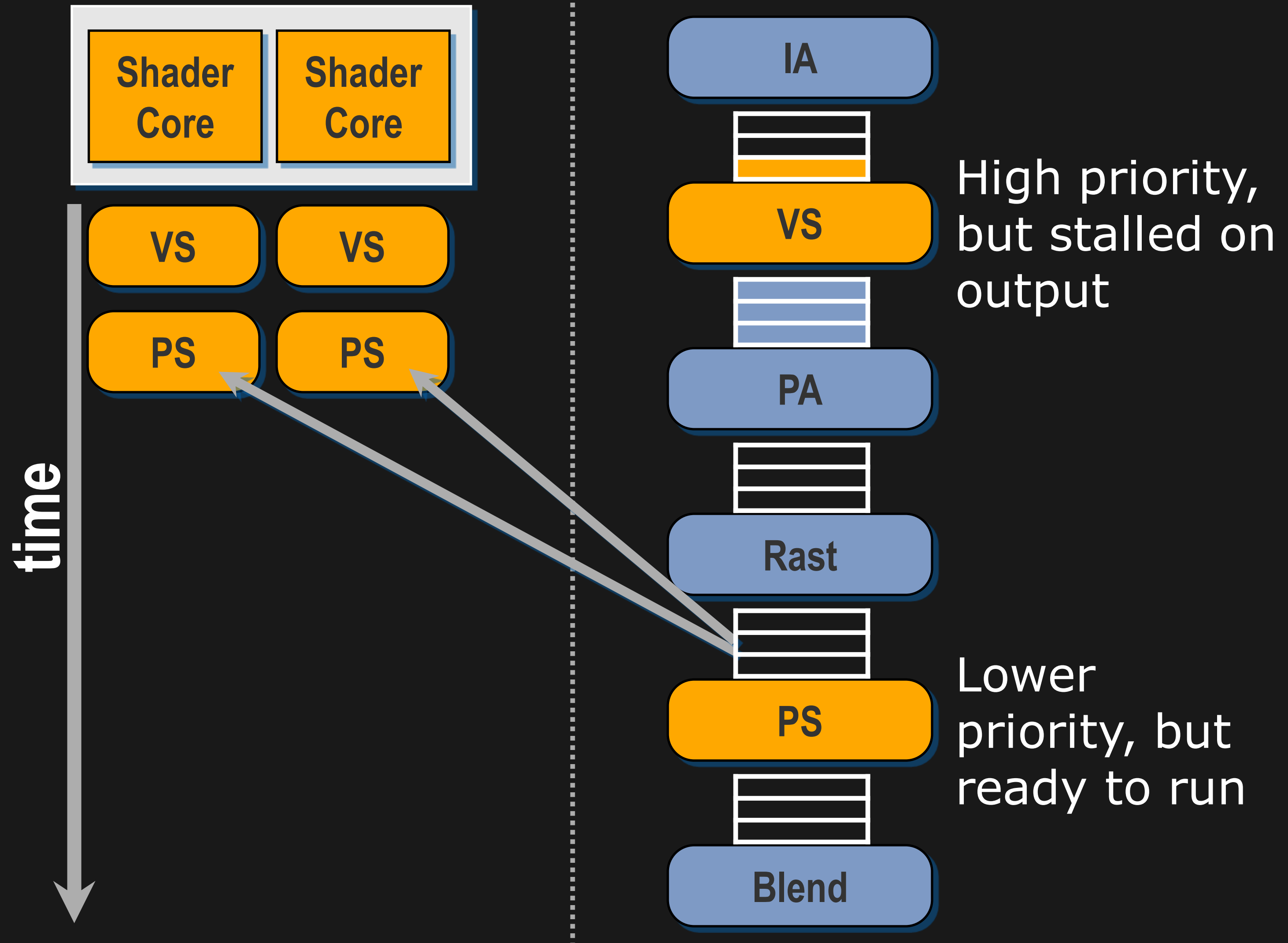
Scheduling the pipeline



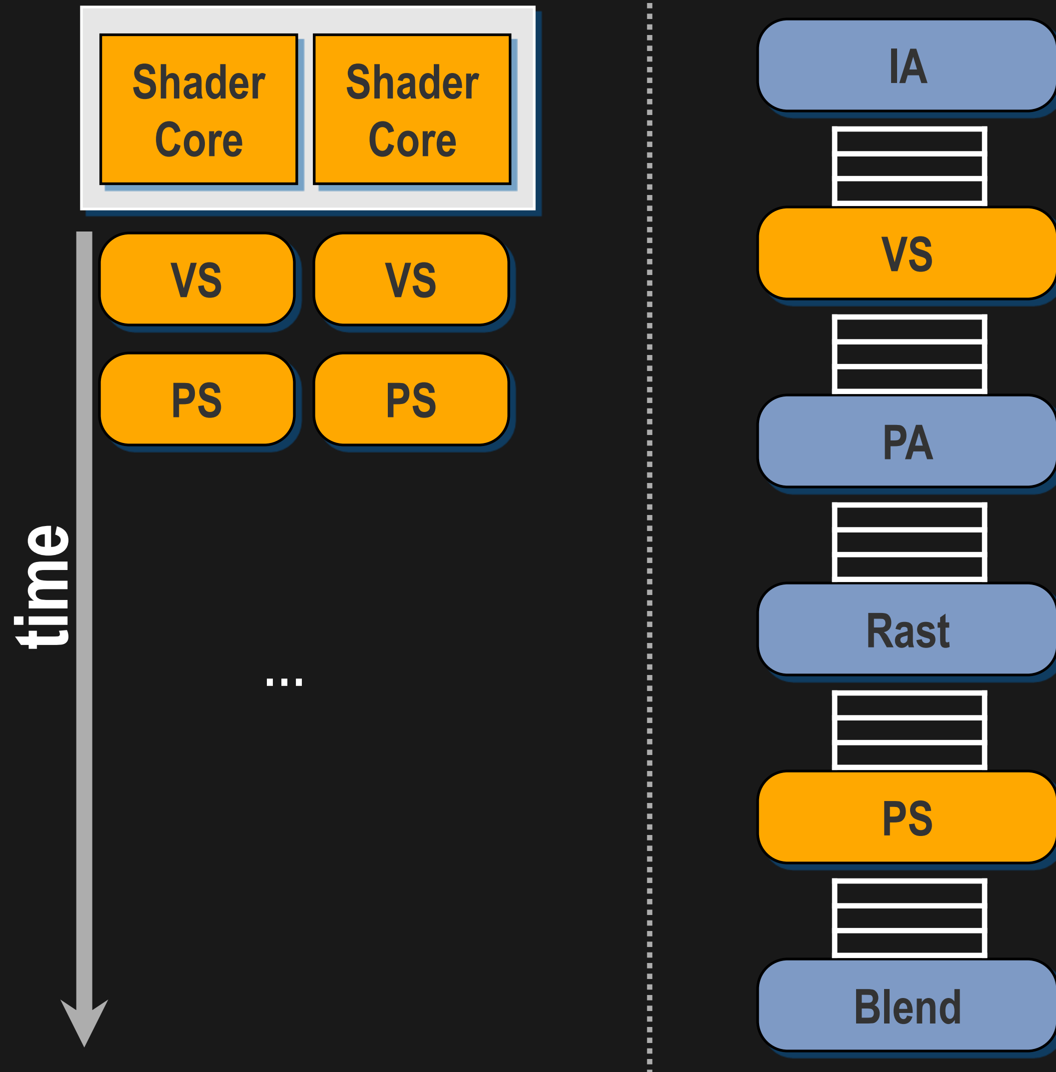
Scheduling the pipeline



Scheduling the pipeline



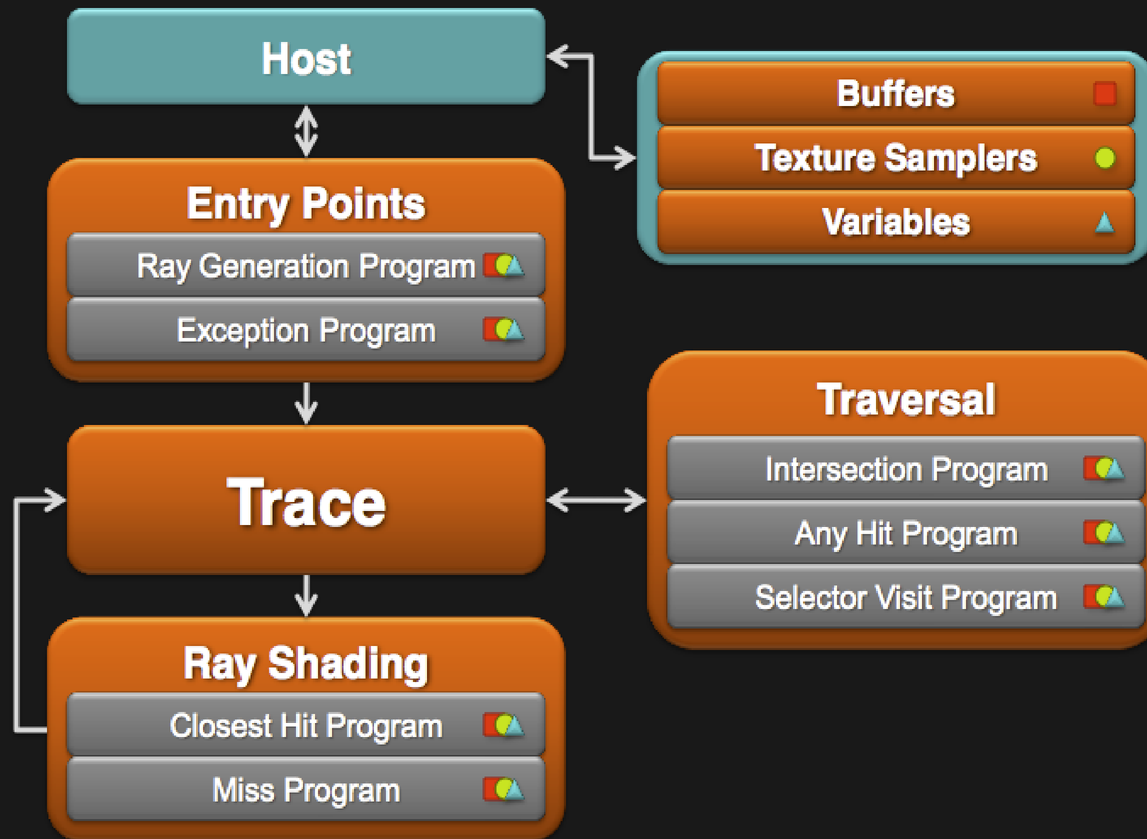
Scheduling the pipeline



Scheduling the pipeline

Queue sizes and backpressure provide a natural knob for balancing **horizontal batch coherence** and **producer-consumer locality**.

OptiX



Summary

Key concepts

- Think of scheduling the pipeline as mapping tasks onto cores.
- Preallocate resources before launching a task.
  - Preallocation helps ensure forward progress and prevent deadlock.
- Graphics is irregular.
  - Dynamically generating, aggregating and redistributing tasks at irregular amplification points regains coherence and load balance.
- Order matters.
  - Carefully structure task redistribution to maintain ordering.

Why don’t we have dynamic resource allocation?
e.g. recursion, malloc() in shaders

Static preallocation of resources guarantees forward progress.

Tasks which outgrow available resources can stall, causing **deadlock**.

Geometry Shaders are slow because they allow dynamic amplification in shaders.

- Pick your poison:
  - **Always stream through DRAM.**
    - *Exemplar: ATI R600*
    - Smooth falloff for large amplification, but very slow for small amplification (DRAM latency).
  - **Scale down parallelism to fit.**
    - *exemplar: NVIDIA G80*
    - Fast for small amplification, poor shader throughput (no parallelism) for large amplification.

Why isn’t rasterization programmable?

Yes, partly because it is computationally intensive, *but also:*

It is *highly irregular*.

- It must generate and aggregate *regular output*.
- It must integrate with an *order-preserving task redistribution mechanism*.

Questions for the future

Can we **relax** the **strict ordering** requirements?

Can you build a **generic scheduler** for **application-defined pipelines**?

What **application-specific information** would a **generic scheduler** need to work well?

Starting points to learn more

The next step: parallel primitive processing

- Eldridge et al. *Pomegranate: A Fully Scalable Graphics Architecture*. SIGGRAPH 2000.
- Tim Purcell. *Fast Tessellated Rendering on Fermi GF100*. Hot3D, HPG 2010.

Scheduling cyclic graphs, in software, on current GPUs

- Parker et al. *OptiX: A General Purpose Ray Tracing Engine*. SIGGRAPH 2010.

Details of the ARM Mali design

- Tom Olson. Mali-400 MP: A Scalable GPU for Mobile Devices. Hot3D, HPG 2010.

“Cypress” Deep Dive

ATI Radeon™ HD 5870 GPU Features

	ATI Radeon™ HD 4870	ATI Radeon™ HD 5870	Difference
Area	263 mm^2	334 mm^2	1.27x
Transistors	956 million	2.15 billion	2.25x
Memory Bandwidth	115 GB/sec	153 GB/sec	1.33x
L2-L1 Rd Bandwidth	512 bytes/clk	512 bytes/clk	1x
L1 Bandwidth	640 bytes/clk	1280 bytes/clk	2x
Vector GPR	2.62 Mbytes	5.24 MByte	2x
LDS Memory	160 kb	640kb	4x
LDS Bandwidth	640 byte/clk	2560 bytes/clk	4x
Concurrent Threads	15872	31744	2x
Shader (ALU units)	800	1600	2x
Board Power*			
Idle	90 W	27 W	0.3x
Max	160 W	188 W	1.17x

20 SIMD engines (MPMD)

- Each with 16 Stream Cores (SC)
  - Each SC with 5 Processing Elements (PE) (1600 Total)
  - Each PE IEEE 754 –2008 precision capable
    - denorm support, fma, flags, round modes
  - Fast integer support
- Each with local data share memory
  - 32 kb shared low latency memory
  - 32 banks with hardware conflict management
  - 32 integer atomic units

80 Read Address Probes

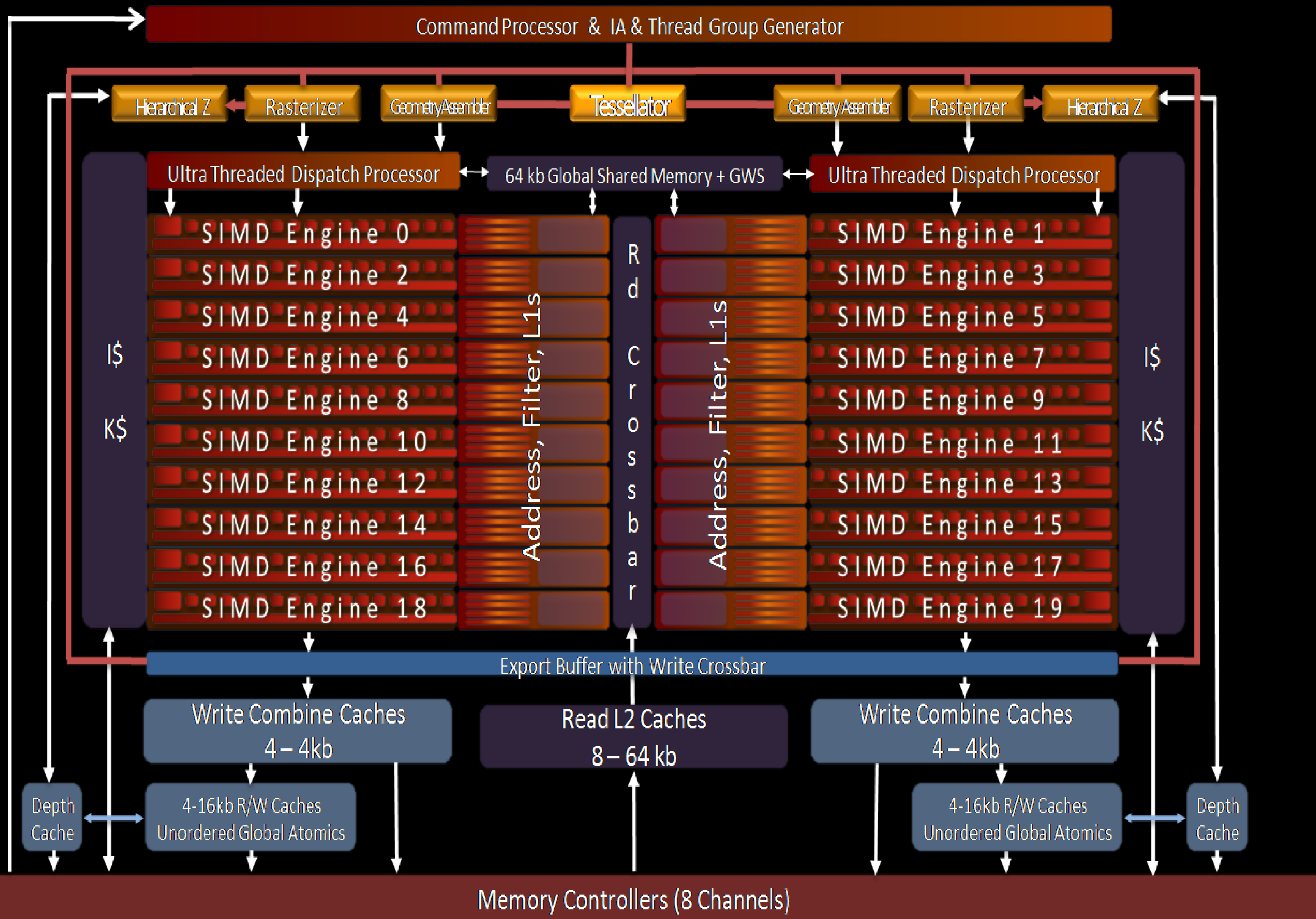
- 4 addresses per SIMD engine (32 -128 bits data)
- 4 filter or convert logic per SIMD

Global Memory access

- 32 SC access read/write/integer atomic/clock
- Relaxed Unordered Memory Consistency Model
- On chip 64 kb global data share

153 GB/sec GDDR5 memory interface

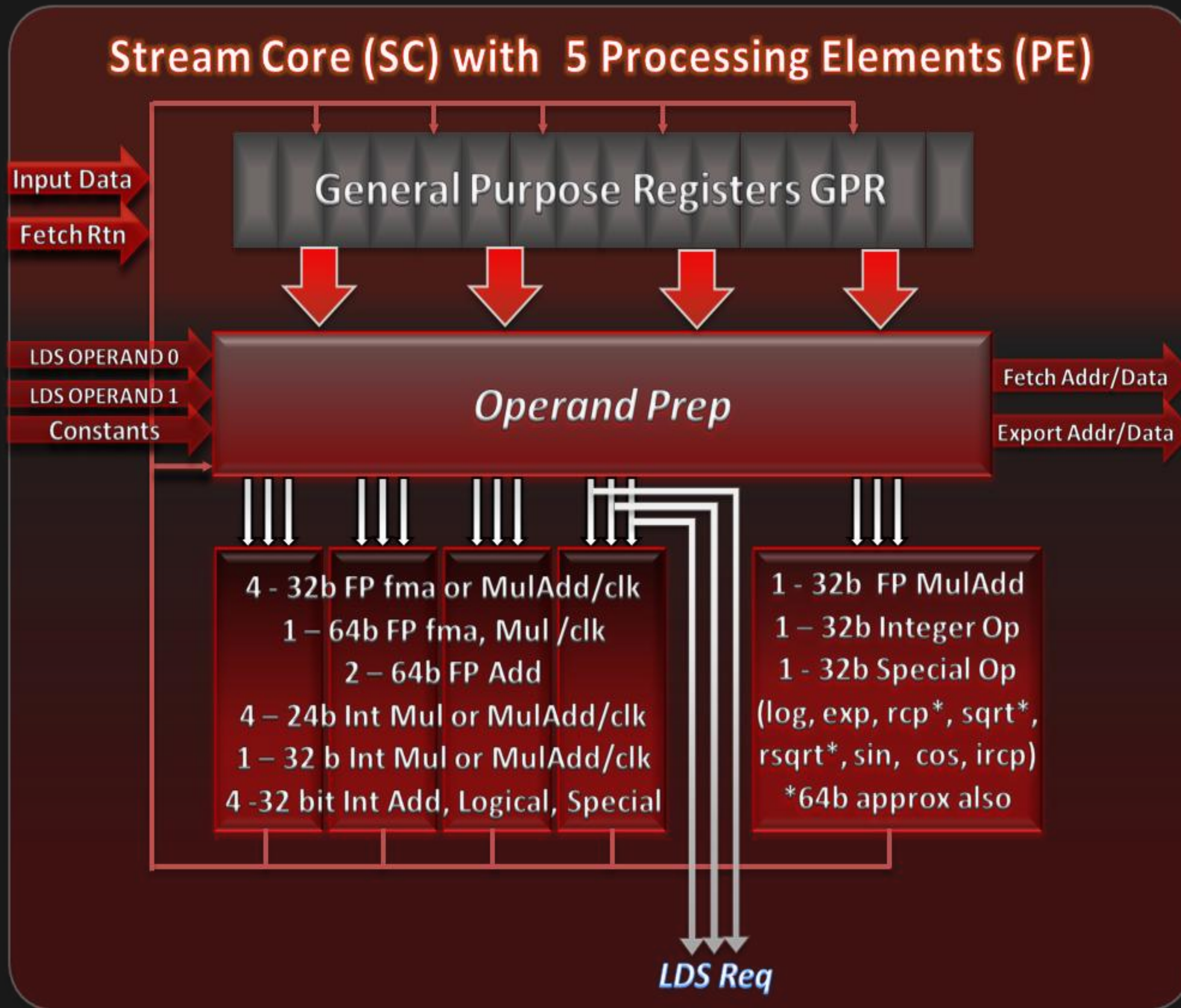
Tera Scale 2 Architecture Radeon™ HD 5870



Compute Aspects of Radeon™ HD 5870

- Stream Cores
- Local Data Share (LDS)
- SIMD Engine
- Load / Store / Atomic Data Access
- Dispatch / Indirect Dispatch
- Global Data Share (GDS)

Stream Core with Processing Elements (PE)



Each Stream Core Unit includes:

- 4 PE
  - 4 Independent SP or Integer Ops
  - 2 DP add or dependant SP pairs
  - 1 DP fma or mult or SP dp4
- 1 Special Function PE
  - 1 SP or Integer Operation
  - SP or DP Transcendental Ops
- Operand Prep logic
- General Purpose Registers
- Data forwarding and predication logic

Processing Element (PE) Precision Improvements

- **FMA** (Fused Multiply Add), IEEE 754-2008 precise with all round modes, proper handling of Nan/Inf/Zero and full de-normal support in hardware for SP and DP
- **MULADD** instruction without truncation, enabling a MUL_{ieee} followed ADD_{ieee} to be combined with round and normalization after both multiplication and subsequent addition.
- **IEEE Rounding Modes** (Round to nearest even, Round toward +Infinity, Round toward –Infinity, Round toward zero) supported under program control anywhere in the shader. Double and single precision modes are controlled separately. Applies to all slots in a VLIW.
- **De-normal Programmable Mode** control for SP and DP independently. Separate control for input flush to zero and underflow flush to zero.
- **FP Conversion Ops** between 16 bit, 32 bit, and 64 bit floats with full IEEE 754 precision.
- **Exceptions Detection** in hardware for floating point numbers with software recording and reporting mechanism. Inexact, Underflow, Overflow, division by zero, de-normal, invalid operation
- **64 bit Transcendental Approximation** Hardware based double precision approximation for reciprocal, reciprocal square root and square root

Processing Element (PE) Improved IPC

- Co-issue of dependant Ops in "ONE VLIW" instruction
  - full IEEE intermediate rounding & normalization
  - Dot4 (A=A*B + C*D + E*F + G*H),
  - Dual Dot2 (A= A*B + C*D; F = G*h + I*J)
  - Dual Dependant Multiplies (A = A * B * C ; F = G * H * I;)
  - Dual Dependant Adds (A = B + C + D; E = F + G + H;)
  - Dependant Muladd (A= A*B+C + D*E; F = G*H + I + J*K)

- 24 bit integer
  - MUL, MULADD (4 – co-issue)
  - Heavy use for Integer thread group address calculation

Processing Element (PE) New Integer Ops

- 32b operand Count Bits Set
- 64b operand Count Bits Set
- Insert Bit field
- Extract Bit Field
- Find first Bit (high, low, signed high)
- Reverse bits
- Extended Integer Math
  - Integer Add with carry
  - Integer Subtract with borrow
- 1 bit pre-fix sum on 64b mask. (useful for compaction)
- Shader Accessible 64 bit counter
- Uniform indexing of constants

Processing Element (PE) Special Ops

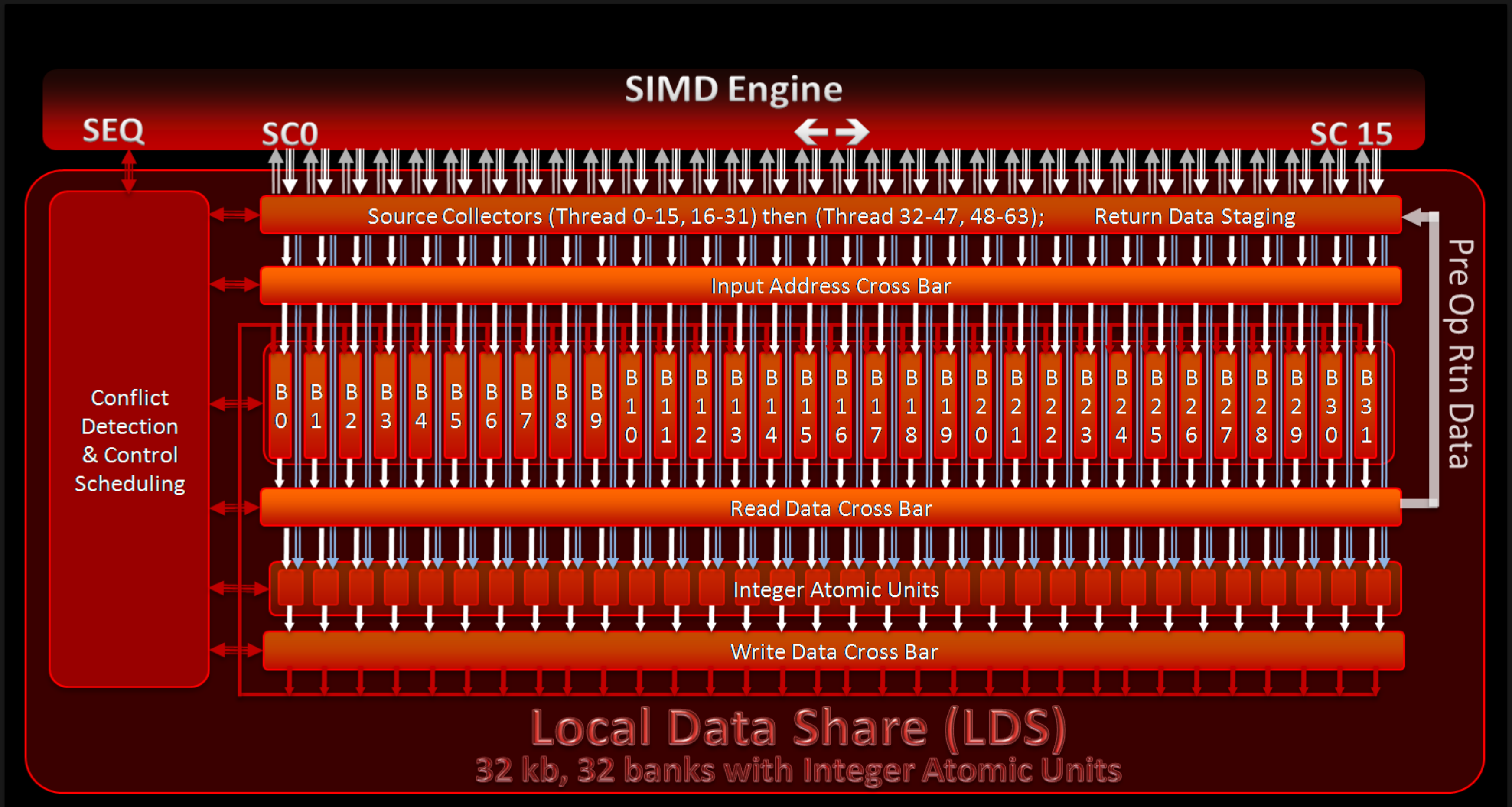
- Conversion Ops
  - FP32 to FP64 and FP64 to FP32 (w/IEEE conversion rules)
  - FP32 to FP16 and FP 16 to FP32 (w/IEEE conversion rules)
  - FP32 to Int/UInt and Uint/Int to FP32
- Very Fast 8 bit Sum of Absolute Differences (SAD)
  - 4x1 SAD per lane, with 4x4 8 bit SAD in one VLIW
  - Used for video encoding, computer vision
  - Exposed via OpenCL extension
- Video Ops
  - 8 bit packed to float and float to 8 bit packed conversion Ops
  - 4 8 bit pixel average (bilinear interpolation with programmable round)
  - Arbitrary Byte or Bit extraction from 64 bits

Local Data Share (LDS)

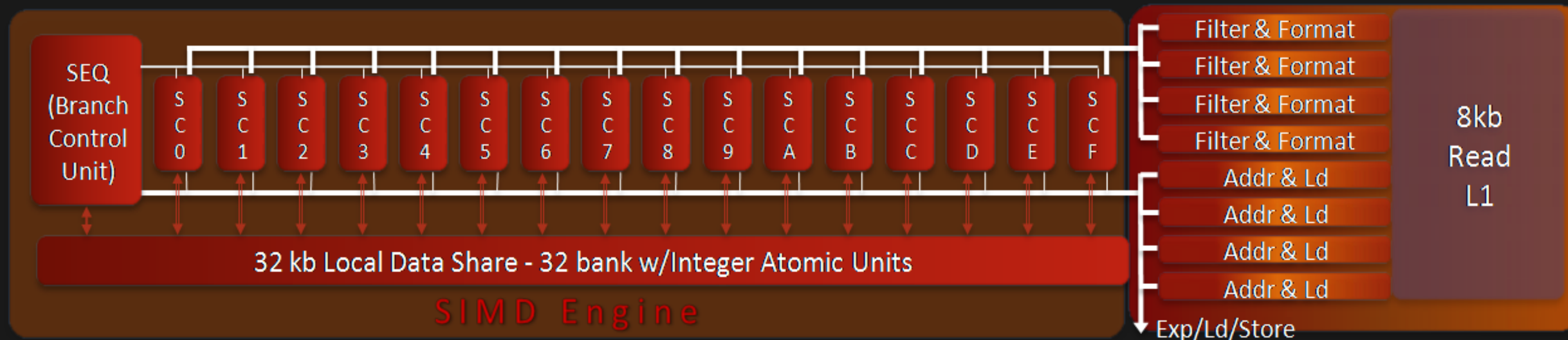
Share Data between Work Items of a Work Group to increase performance

- High Bandwidth access per SIMD Engine – Peak is double external R/W bandwidth
  - Full coalesce R/W/Atomic with optimization for broadcast reads access
- Low Latency Access per SIMD Engine
  - 0 latency direct reads (Conflict free or Broadcast)
  - 1 VLIW instruction latency for LDS indirect Op
- All bank conflicts are hardware detected and serialized as necessary with fast support for broadcast reads
- Hardware allocation of LDS space per thread group dispatch
  - Base and size stored with wavefront for private access
  - Hardware range check - Out of bound access attempts will return 0 and kill writes
- 32 – byte, ubyte, short, ushort reads/writes per clk (reads are sign extended)
- 32 dwords access per clock
  - Per lane 32 bit Read, Read2, Write, Write2
  - Per lane 32 bit Atomic: add, sub, rsub, inc, dec, min, max, and, or, xor, mskor, exchange, exchange2, compare _swap(int), compare_swap(float w/Nan/Inf)
  - Return pre-Op value to Stream Core 4 primary Processing Elements

Local Data Share (LDS)



SIMD Engine



- SIMD Engine can process Wavefronts from multiple kernels concurrently
- Masking and/or Branching is used to enable thread divergence within a Wavefront
  - Enabling each Thread in a Wavefront to traverse a unique program execution path
- Full hardware barrier support for up to 8 Work Groups per SIMD Engine (for thread data sharing)
- Each Stream Core receives up to the following per VLIW instruction issue
  - 5 unique ALU Ops - or - 4 unique ALU Ops with a LDS Op (Up to 3 operands per thread)
- LDS and Global Memory access for byte, ubyte, short, ushort reads/writes supported at 32bit dword rates
- Private Loads and read only texture reads via Read Cache
- Unordered shared consistent loads/stores/atomics via R/W Cache
- Wavefront length of 64 threads where each thread executes a 5 way VLIW Instruction each issue
  - ¼ Wavelength (16 threads) on each clock of 4 clocks (T0-15, T16-31, T32-47, T48-T63)

Radeon™ HD 5870 Compute