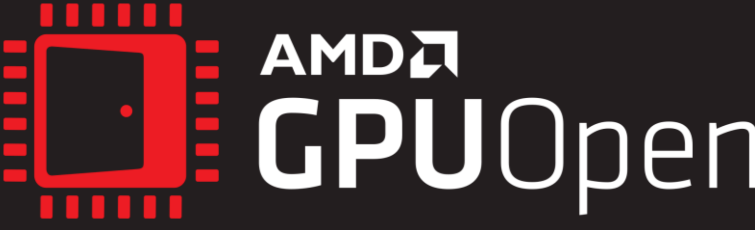
CONCURRENCY MODEL IN EXPLICIT GRAPHICS APIS

DOMINIK BAUMEISTER & DR. MATTHÄUS G. CHAJDAS



WHO WE ARE

Dr. Matthäus G. Chajdas

5 years at AMD

Developer Technology Architect

Dominik Baumeister

3 years at AMD

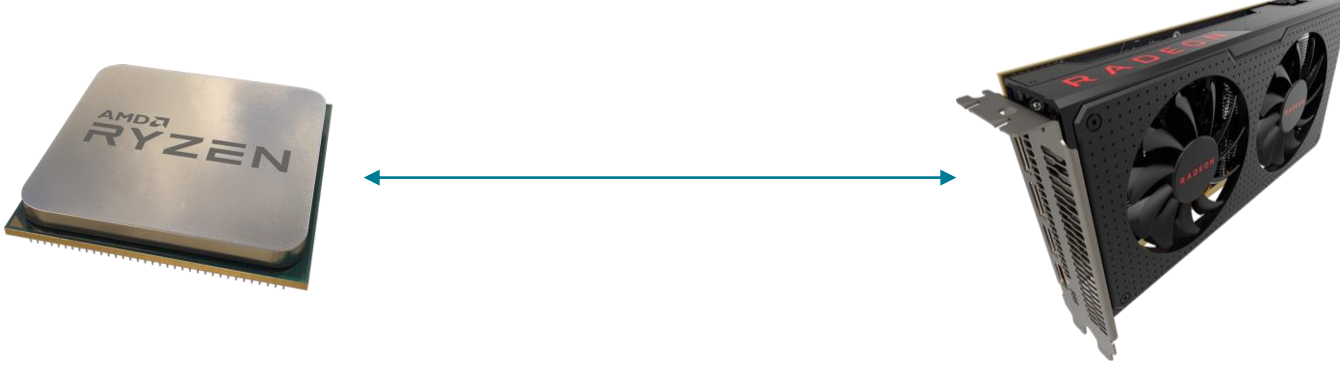
Developer Technology Engineer

GOALS

- Give a (slightly) more in-depth look for people interested in graphics programming
- Prepare the mental model for explicit graphics APIs (DirectX12®, Vulkan®)

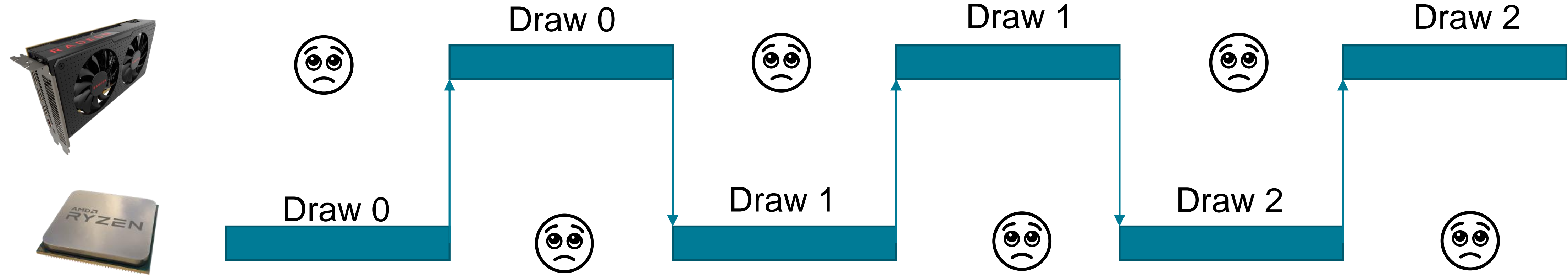
BASICS

- Our program executes on the CPU
- Shaders can run as part of a Draw() or Dispatch() execution on the GPU
- CPU and GPU are physically separate entities (even in case of an integrated GPU, they‘re separate blocks on the chip)



NAIVE VIEW

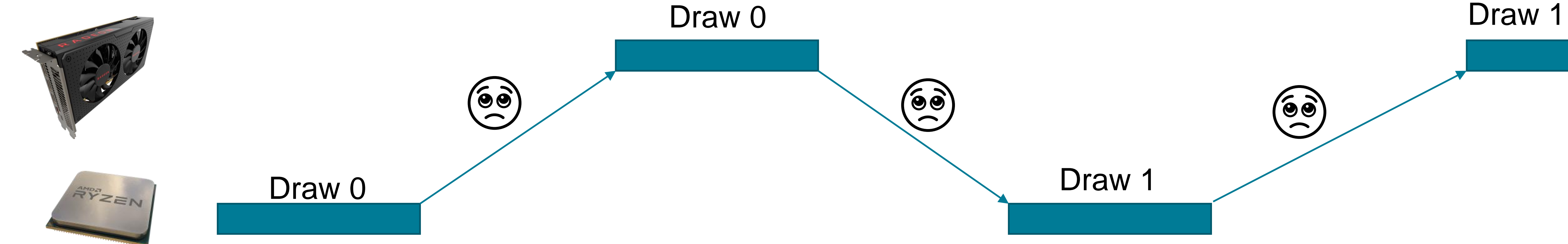
- What would happen if a `Draw()` would be executed immediately on the GPU (i.e., like calling a function)



- Of course not what we would like to see

NAIVE VIEW

- In addition, „immediately“ is actually „quite some time later“ in practice as the commands have to be sent via a message across the bus



COMMAND RECORDING

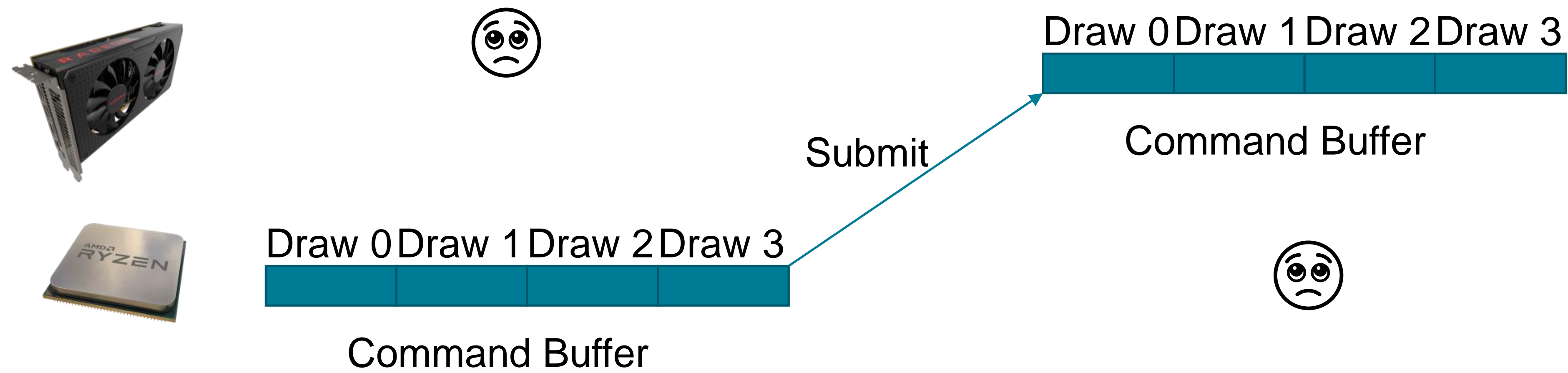
- Instead: Keep most of the commands somewhere in memory
- Accumulate a lot of them (ideally enough to keep the GPU busy for a while)
- Then „Submit“ them to the GPU

Draw 0	Draw 1	Draw 2	Draw 3

Command Buffer

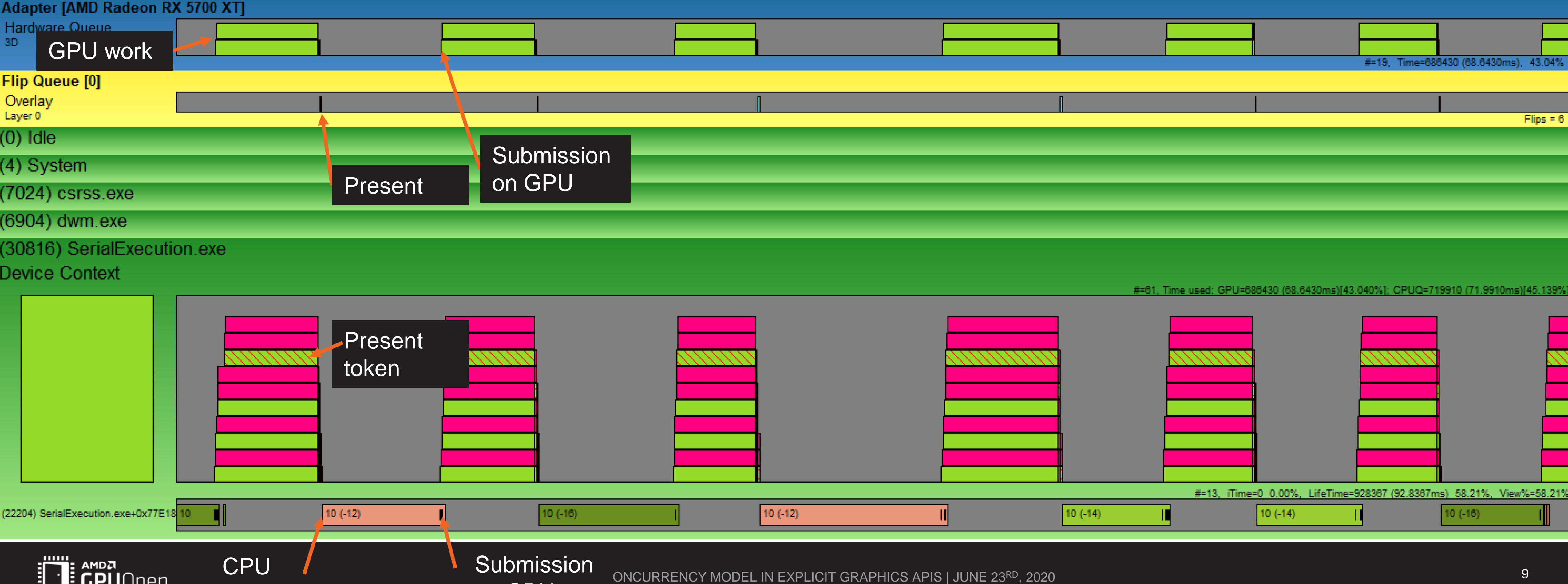
COMMAND RECORDING

- Again, we‘re running into the same challenge (conceptually):
  - CPU is working hard to create the commands, meanwhile the GPU is idle
  - Then GPU is working hard to execute the commands, meanwhile the CPU is idle



SERIAL EXECUTION

- How can I detect these cases? – GPUView (https://graphics.stanford.edu/~mdfisher/GPUView.html)

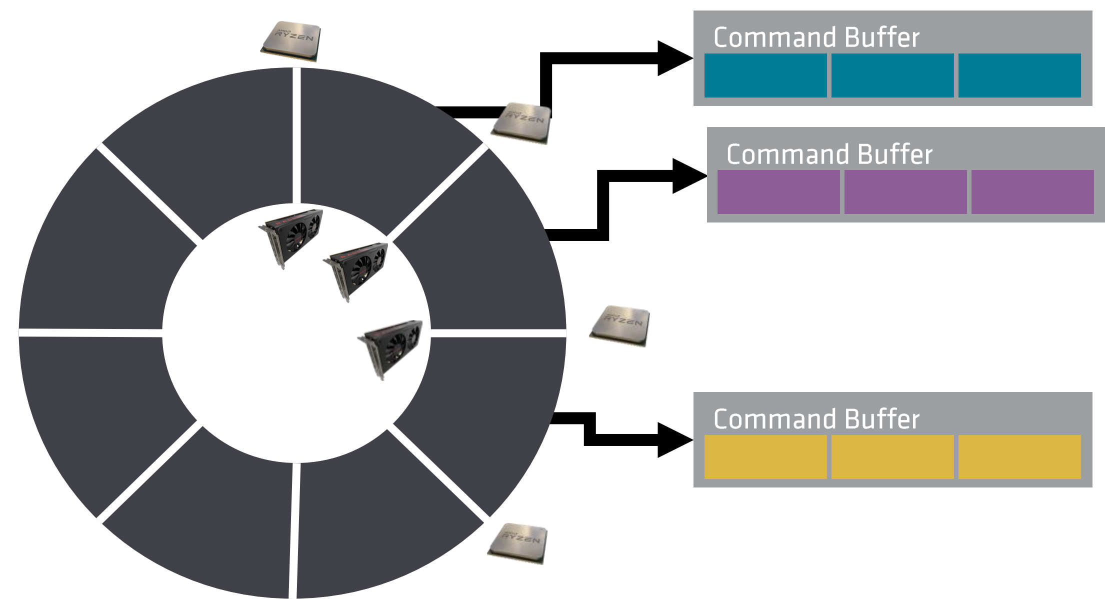


SERIAL EXECUTION

- Reminder: GPU and CPU are separate entities with their own timeline and resources. They don‘t necessarily run in sync!
- This means in practice you actually have to do additional work to sync them up (and thus to generate the pathological case shown before ☺)

RING BUFFER

- The CPU tells the GPU where to find the commands in memory via a ring buffer
- The CPU can then move on to do some meaningful work while GPU executes the commands
- Possibly the CPU is submitting more command buffers, even though the GPU is still churning on the first one

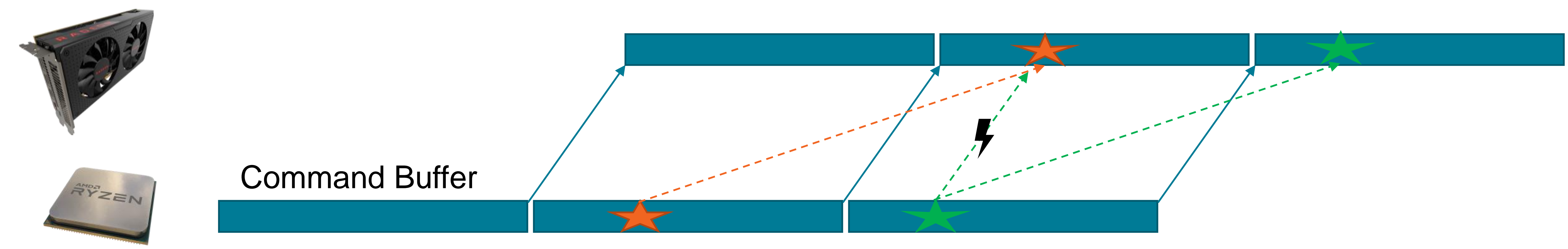


OVERLAPPING EXECUTION

- In GPUView (https://graphics.stanford.edu/~mdfisher/GPUView.html)

Adapter [AMD Radeon RX 5700 XT]
Hardware Queue
3D
#=33, Time=948121 (94.8121ms), 74.52%
Flip Queue [0]
Overlay
Layer 0
Flips = 11
(0) Idle
(4) System
(7024) csrss.exe
(6904) dwm.exe
(29432) OverlappingExecution.exe
Device Context
#=99, Time used: GPU=948121 (94.8121ms)[74.525%]; CPUQ=1002714 (100.2714ms)[78.816%]
#=1, iTime=0 0.00%, LifeTime=1272223 (127.2223ms) 100.00%, View%=100.00%
(8116) OverlappingExecution.exe+0x77E0(812)

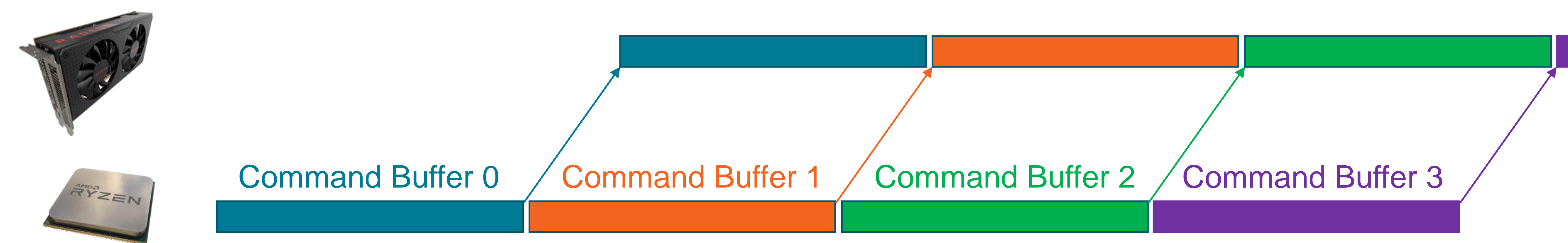
OVERLAPPING EXECUTION



- Benefits: GPU and CPU can work in parallel
- Drawbacks:
  - Can‘t overwrite command buffers that are already in flight
  - Can‘t overwrite constants that are already in flight
  - Can‘t change/destroy textures and buffers that are currently in use

BUFFERING

- The solution is buffering



BUFFERING

- Keep multiple copies of your
  - Command buffers
  - Descriptors
    - Small objects that describe your buffers & textures
    - Memory address, size, mip level count, swizzle …
  - Constant buffers
- Most of that is done for you on older APIs
- On recent, explicit APIs you are responsible to do that yourself

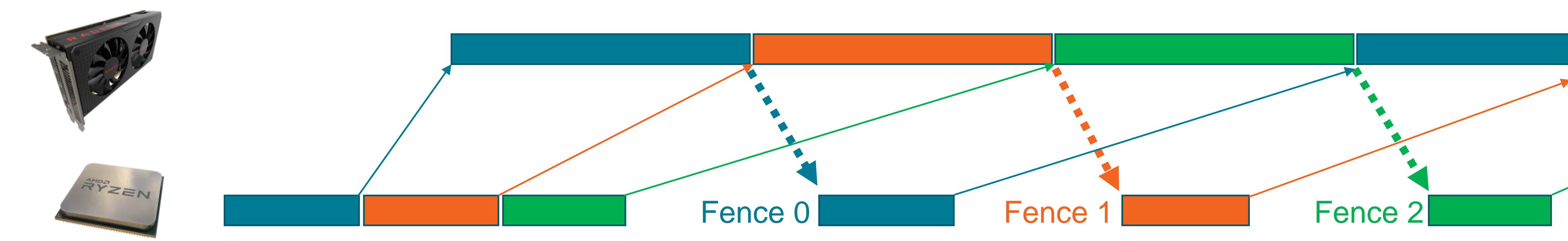
BUFFERING

- Now suppose our CPU is really really fast or in more general terms:
  Executing work on the GPU
  takes a lot longer than on the CPU
- GPU View – again
  (https://graphics.stanford.edu/~mdfisher/GPUView.html)
- Tripple (quadruple, …) buffering?
  - Can help with smoother experience
  - At the expense of latency
    = time from user input to image on screen
  - Can‘t do this forever
  - Usually games do double or triple buffering

Adapter [AMD Radeon RX 5700 XT]
Hardware Queue
3D
Flip Queue [0]
Overlay
Layer 0
(0) Idle
(4) System
(7024) csrss.exe
(6904) dwm.exe
(29848) MultipleFramesInFlight.exe
Device Context
No Gaps
Queued
Frames
(30416) MultipleFramesInFlight.exe+0x77E08
10
10
11
11
11

SYNCHRONIZATION - FENCES

- At some point we need to know when the GPU has finished executing some work
- This is done via **Fences**
- If done correctly – ~ once a frame – this does not cause serial bottlenecks

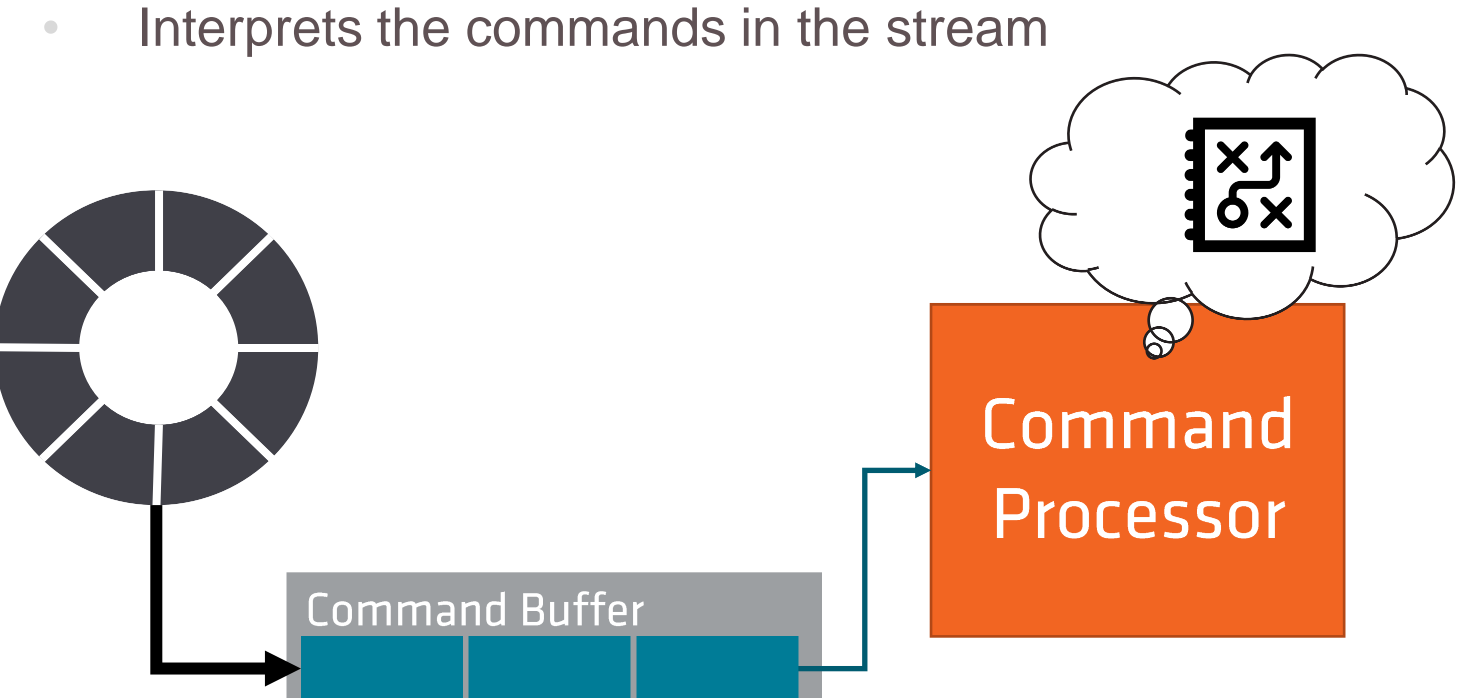


READBACK

- Now: What should we do if we want to read data from GPU memory on the CPU?
  - Screenshots
  - Timestamps for profiling
  - Occlusion queries (ever wondered where the white/black flashes in some games come from?)
- We don‘t want to sync with the latest frame, this would bring us back to square one = CPU and GPU running serialized
- Instead, the CPU should continue performing useful work and periodically poll the GPU when the frame is complete

COMMAND PROCESSOR

- Leaving the realm of the CPU, we‘re now **on the GPU**
- Reads the pointer to the command buffer from the ring buffer
- Interprets the commands in the stream



COMMAND PROCESSOR

- Schedules them for the respective queues
  - There are Graphics Queues
  - And Compute Queues
- More parallelism!

QUEUES

- Different synchronization domains
- But share the same ALUs, registers
- For that reason they are useful to overlap ALU heavy work with fixed function heavy work
- Two ALU limited workloads on different queues will likely not run faster than on a single queue
- But for example shadow map creation (rasterizer limited) and ambient occlusion (ALU limited) go hand in hand pretty well

QUEUES

START | OVERVIEW | EVENTS | SETTINGS

- Frame summary
- Barriers
- Most expensive events
- Context rolls
- Render/depth targets
- Device configuration

System activity

Visualize command buffer submission and synchronization primitives.

GPU-based frames | Workload views | CPU submission markers | Zoom to selection | Reset zoom

- [ ] Command buffers
- [x] Sync objects
- [ ] Sequential
- [ ] GPU only

Frame 698 | Frame 699 | Detailed GPU events | Frame 701 | Frame 702

25.000 ms | 30.000 ms | 32.500 ms | 35.000 ms | 37.500 ms | 40.000 ms | 42.500 ms | 45.000 ms | 47.500 ms | 50.000 ms | 52.500 ms

Graphics queue

vkQueuePresentKHR()
vkSemaphoreWait(0x12ae33afef0)
vkSemaphoreWait(0x12ad7afc5e0)
vkSemaphoreSignal(0x12ae33afef0)
vkSemaphoreSignal(0x12ad7afc5e0)
QueueSubmit(2)
vkSemaphoreWait(0x12ae33b0910)
QueueSubmit(4)

Compute queue

vkSemaphoreSignal(0x12ae33b1330)
QueueSubmit(1)
vkSemaphoreSignal(0x12ae33b0910)
vkSemaphoreWait(0x12ae33b0c70)
vkSemaphoreWait(0x12ae33b18d0)

Graphics | Compute | DMA | Wait | Signal | Present

Submit time: 32.934 ms | Submit duration: 10.803 ms | Enqueue duration: 10.803 ms | GPU duration: 0.000 ms | 28.368 ms

1010
1010

SEMAPHORES

- Again, somehow have to make sure that two queues can be synchronized:
  Done using **Semaphores (Vulkan)** or **Fences (D3D12).**
- Semaphores can be signalled and waited on
- They have some overhead – both on CPU and GPU
  - GPU has to fill up again (from the very front of the pipeline)
  - OS on the CPU makes sure that you cannot deadlock the GPU
    i.e., avoid that one application can make the whole system unresponsive

SEMAPHORES

Frame 5903 | Detailed GPU events | Frame 5905

000 ms | 53.000 ms | 54.000 ms | 55.000 ms | 56.000 ms | 57.000 ms | 58.000 ms | 59.000 ms | 60.000 ms | 61.000 ms | 62.000 ms | 63.000 ms | 64.000 ms | 65.000 ms | 66.000 ms | 67.000 ms | 68.000

Graphics queue

[0x17eb2d72dd0] VkSemaphoreSignal
[248577] VkCommandBuffer
[0x17eb2d72830] VkSemaphoreWait
[0x17e33be96f0] VkSemaphoreWait
[0x17e33bea110] VkSemaphoreSignal
vkQueuePresentKHR
[0x17eb2d72b90] VkSemaphoreWait
[0x17eb2d72b90] VkSemaphoreSignal
[0x17eb2d72950] VkSemaphoreWait

Compute queue

[0x17e33be96f0] VkSemaphoreSignal
[242689] VkCommandBuffer
[0x17e33bea110] VkSemaphoreWait
[246017] VkCommandBuffer

Graphics | Compute | DMA | Wait | Signal | Present

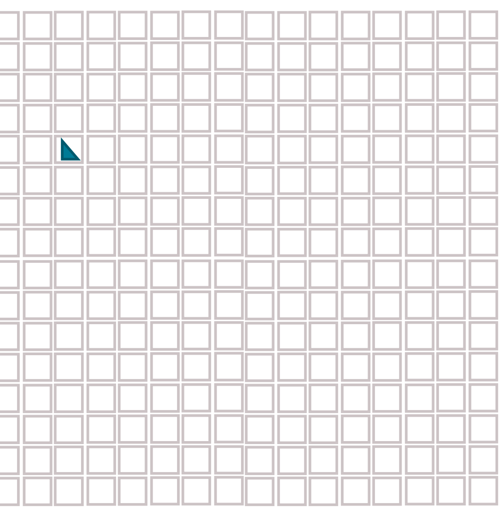
Submit time: 50.685 ms | Submit duration: 6.335 ms | Enqueue duration: 5.135 ms | GPU duration: 1.200 ms | 52.047 ms

SHADER CORE

- Now to the actual execution in the shader core
- Loads of vector units (execute in lock step)
- Bunched up in Compute Units (execute independently from each other)
- „Dual Compute Unit“ – Radeon™ RX 5700 XT contains 20 of those → 2560 hardware threads in total

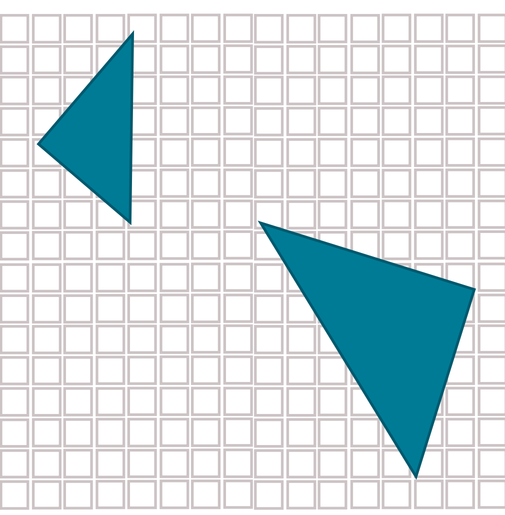
SHADER CORE

- For full utilization we want to avoid synchronous execution as much as possible.
- Imaging a huge GPU with thousands of ALUs and then your triangles cover a single pixel. 99% of the GPU would idle.



PARALLEL EXECUTION

- APIs require draw calls to be issued in the order they have been submitted
- They also require the results to become visible in the order they have been submitted
- GPUs go to great lengths to execute in parallel while maintaining the illusion of serial execution
  - This can cause interesting (correct) behavior.
  - Imagine two triangles not overlapping:
    The later one can finish before the first.



BARRIERS

- Stalls execution until all remaining work **on the same queue** has been done.

Number of workitems in flight (Occupancy)

Barriers

PS: This is bad practice, FidelityFX SPD (https://gpuopen.com/fidelityfx-spd/) shows how to do it better.

BARRIERS

- Handle visibility operations
  i.e., possibly insert cache flushes to make contents of L0 caches visible to other compute units

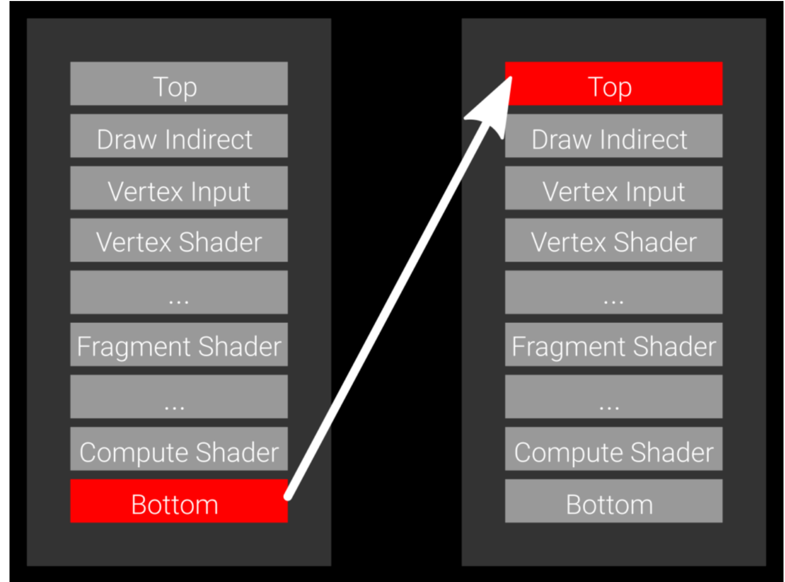
Dual Compute Unit
L0$ I$ and K$ L0$

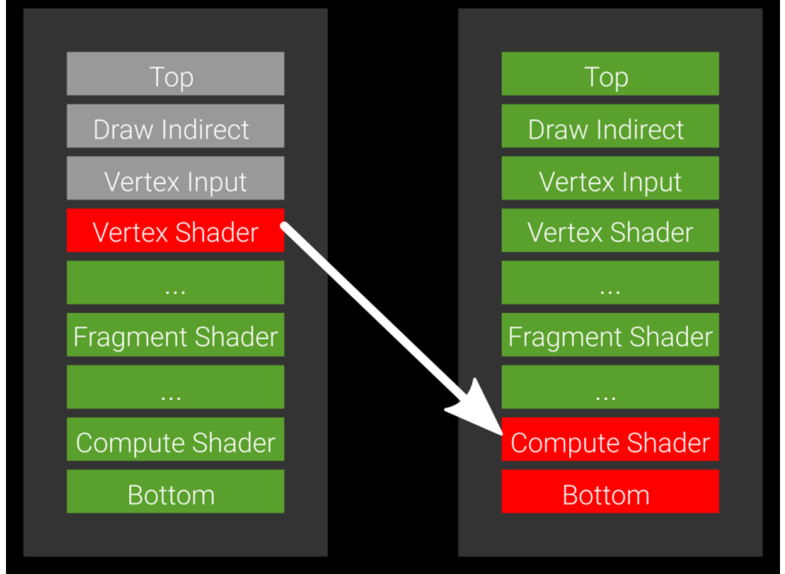
L1$
L1$
L1$
L1$
L2$

Cache Hierarchy Radeon™ RX 5700 XT (simplified)

BARRIERS

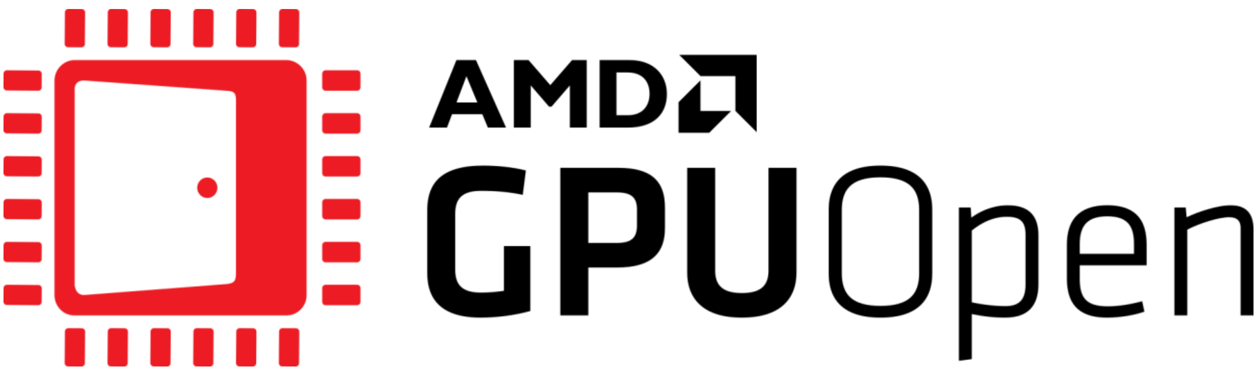
- There is more parallelism: the Graphics Pipeline is well… a pipeline
- The different stages for a Draw/Dispatch run (partly) in serial,
  but new work can already start while old work items are still being executed
- More on **GPUOpen**: https://gpuopen.com/learn/vulkan-barriers-explained/





FURTHER READS

- Lots of relevant information on **GPUOpen** https://gpuopen.com
- Including practical tools like the **Radeon GPU Profiler** https://gpuopen.com/rgp/
- **RDNA** https://gpuopen.com/documentation/



- Keeping your GPU fed without getting bitten https://www.khronos.org/assets/uploads/developers/library/2017-khronos-uk-vulkanised/004-Synchronization-Keeping%20Your%20Device%20Fed_May17.pdf
- Breaking down Barriers https://therealmjp.github.io/posts/breaking-down-barriers-part-1-whats-a-barrier/
- A trip through the graphics pipeline https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/
- GPU View https://graphics.stanford.edu/~mdfisher/GPUView.html

DISCLAIMER

DISCLAIMERS

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

Use of third-party marks / products is for informational purposes only and no endorsement of or by AMD is intended or implied. GD-83

AMD Radeon FreeSync requires a monitor and AMD Radeon™ graphics, both with FreeSync support. Seewww.amd.com/freesync for complete details. Confirm capability with your system manufacturer before purchase. GD-127

© 2020 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Radeon and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Windows and DirectX are registered trademarks of Microsoft Corporation in the US and other jurisdictions. Vulkan and the Vulkan logo are trademarks of Khronos Group Inc. Other names are for informational purposes only and may be trademarks of their respective owners.