Breaking Down Barriers: An Intro to GPU Synchronization

Matt Pettineo
Lead Engine Programmer
Ready At Dawn Studios
Who am I?

- Ready At Dawn for 9 years
  - Lead Engine Programmer for 5
- I like GPUs and APIs!
- Lots of blogging, Twitter, and GitHub
  - You may know me as MJP!
What is this talk about?

- GPU Synchronization!
- What is it?
- Why do you need it?
- How does it work?
- How does it affect performance?
Barriers in D3D12/Vulkan

- New concept!
- Annoying
  - D3D11 didn’t need them!
- Difficult
  - People keep talking about them
- Affects performance
  - But why? And how?
CPU Thread Barriers

- Thread sync point
- “Wait until all threads get here”
  - Spin wait
  - OS primitives
- Barrier is a toll plaza
CPU Memory Barriers

- Ensure correct order of reads/writes
  - Ex: write finishes before barrier, read happens after
- Affects CPU memory ops
  - and compiler ordering!
- Barrier is a doggie gate
What’s The Common Thread?

- Dependencies!
- Task A produces something
- Task B consumes something
- Task B depends on Task A
- Results need to be **visible** to dependent tasks!
Single-Threaded Dependencies

- int a = GetOffset(); int b = myArray[a];
- The compiler + CPU have your back!
  - Automatic dependency analysis
  - No need for manual barriers
  - Expected ordering on a single core
- Easy mode
Multi-Threaded Dependencies

- Dependencies no longer visible!
  - Arbitrary numbers of threads
  - Free-for all memory access
- CPU mechanisms break down
  - Per-core store buffers and caches
- Everyone has failed you
  - You’re on your own
Task Dependencies

CPU

Core 0
Get Bread

Core 1
Spread Peanut Butter

Tasks Overlap!
Task Dependencies

- Core 0: Get Bread
- Core 1: Peanut Butter

No Overlap!
GPU Parallelism

- GPU is **not** a serial machine!
- Looks are deceiving
- HW and drivers help you out
GPUs are Thread Monsters!
GPUs are Thread Monsters!

- Lots of overlapping when possible
  - No dependencies
  - Re-ordering for render target writes (ROPs)
- Overlap improves performance!
  - More on this later
GPU Thread Barriers

- Dependencies between draw/dispatch/copy
- Wait for batch of threads to finish
  - Same as CPU task scheduler
- Often called “flush”, “drain”, “WaitForIdle”
GPU Cache Barriers

- Lots of caches!
- Not always coherent!
  - Different from CPU’s
- Flush and/or invalidate to ensure **visibility**
- Batch your barriers!

Uh oh
GPU Compression Barriers

- HW-enabled lossless compression
  - Delta Color Compression (DCC)
  - Saves bandwidth
- (may) Decompress for read
- Decompress for UAV write
D3D12 Barriers

- Higher level - “resource state” abstraction
  - Texture is in an SRV read state
  - Buffer is in a UAV write state
  - Mostly describes resource visibility
- Implicit dependencies from state transition
- Layout/compression also implied
Vulkan Barriers

- More explicit (verbose) than D3D12
- Specifies
  - Producing/consuming GPU stage
  - Read/write state
  - Texture layout
D3D12/Vulkan Barriers

- Both abstract away GPU specifics
- Both let you over-sync/flush/decompress
- RGP will show you!
- PIX can warn you!
What about D3D11?

- Driver tracked dependencies!
  - Like a run-time compiler
  - Easy mode
- Lots of CPU work!
- Hard to do multithreaded
- Requires CPU-visible resource binding

Incompatible with D3D12/Vulkan!
Let’s Make a GPU!

Introducing: The MJP-3000
MJP-3000 Limitations

- Compute only
- Only 16 shader cores
- No SIMD
- No thread cycling
- No caches
Simple Dispatch Example

- Dispatch 32 threads
- Each thread writes 1 element to memory
Simple Dispatch Example

DISPATCH(A, 32)

NOP

NOP

0 cy
Simple Dispatch Example

```
DISPATCH(A, 32)
```

0 cy

32 threads enqueued
Simple Dispatch Example
Simple Dispatch Example

Threads write data to memory
Simple Dispatch Example

100 cy

DISPATCH(A, 32)

NOP

NOP

NOP

Remaining threads start executing
Simple Dispatch Example

All threads are done writing to memory
Thread Barrier Example

- Dispatch B is **dependent** on Dispatch A
  - We can’t have any overlap!
- New command: **FLUSH**
  - Command processor waits for thread queue and shader cores to become empty
Thread Barrier Example

- DISPATCH(A, 24)
- FLUSH
- DISPATCH(B, 24)
Thread Barrier Example
Thread Barrier Example

100 cy

- DISPATCH(A, 24)
- FLUSH
- DISPATCH(B, 24)
- NOP

FLUSH waits for queue to empty

No overlap!

Cores are idle!
Thread Barrier Example

200 cy

FLUSH

DISPATCH(B, 24)

NOP

NOP
Thread Barrier Example
Thread Barrier Example
Thread Barrier Example
Thread Barrier Example
Thread Barrier Example

- FLUSH prevented overlap 😊
- ...but cores were 50% idle for 200 cycles
  - 75% overall utilization 😞
  - Took 400 cycles instead of 300 cycles
The Cost of a Barrier

- Barrier cost is relative to the drop in utilization!
- Gain from removing a barrier is relative to % of idle shader cores
- Larger dispatches => better utilization
- Longer running threads => high flush cost
  - Amdahl’s Law
D3D12/Vulkan Barriers are Flushes!

- Expect a thread flush for a transition/pipeline barrier between draws/dispatches
- Same for a D3D12RESOURCE_UAV_BARRIER
- Try to group non-dependent draws/dispatches between barriers
- May not be true for future GPUs!
Overlapping Dispatches Example

- Dispatch B still dependent on Dispatch A
- Dispatch C dependent on neither
- Let’s try to recover some perf from idle cores
Overlapping Dispatches Example
Overlapping Dispatches Example
Overlapping Dispatches Example

Threads from Dispatch C keep our cores busy!
Overlapping Dispatches Example
Overlapping Dispatches Example
Overlapping Dispatches Example
Overlapping Dispatches Example

- Same **latency** for Dispatch A + Dispatch B
  - But we got Dispatch C for free!
  - Overall **throughput** increased
- Saved 100 cycles vs. sequential execution
- 75%->87.5% utilization!
Insights From Overlapping

● What if we think of the GPU as a CPU?
  ● Each command is an instruction
● Overlapping == Instruction Level Parallelism
● Explicit parallelism, not implicit
  ● Similar to VLIW (Very Long Instruction Word)
Bad Overlap Example
Bad Overlap Example

DISPATCH(A, 24)
DISPATCH(C, 8)
FLUSH
DISPATCH(B, 24)
Bad Overlap Example

Uh oh
Bad Overlap Example
Bad Overlap Example
Bad Overlap Example
What Happened?

- 400 cycles with 50% idle cores
  - 71.4% utilization
- 1 CP -> 1 queue -> global flush/sync
  - B wanted to sync on A, but also synced on C
- Re-arranging could help a bit
  - But wouldn’t fix the issue
Why Not *Two* Command Processors?
Upgrading To The MJP-4000

Second Front End
Introducing The MJP-4000

- Two front-ends
  - Two command processors for syncing
  - Two thread queues
  - Two independent command streams
- Still 16 shader cores
  - Max throughput same as MJP-3000
  - First-come first-serve for thread queues
Dual Command Stream Example

- Dispatch A -> 68 threads, 100 cycles
- Dispatch B -> 8 threads, 400 cycles
  - B depends on A
- Dispatch C -> 80 threads, 100 cycles
- Dispatch D -> 80 threads, 100 cycles
  - D depends on C

Independent command streams
Dual Command Stream Example

First command stream submitted

DISPATCH(A, 68)
FLUSH
DISPATCH(B, 8)
Dual Command Stream Example

Second command stream submitted

All cores are busy – threads stay in the queue
Dual Command Stream Example

Cores are free – queues will split available cores
Dual Command Stream Example

- DISPATCH(A, 68)
- DISPATCH(B, 8)
- DISPATCH(C, 80)
- DISPATCH(D, 80)

- FLUSH

- 44
- 72

- 100 cy
Dual Command Stream Example

 Dispatch A has only 4 threads left, but Dispatch C keeps the remaining cores busy!
Dual Command Stream Example

Dispatch B can only saturate half the cores, but Dispatch C can fill the rest!
Dual Command Stream Example

DISPATCH(B, 8) 24
DISPATCH(D, 80)
DISPATCH(C, 80)
DISPATCH(B, 8)
DISPATCH(C, 80)
DISPATCH(D, 80)

FLUSH
FLUSH
FLUSH

900 cy
Dual Command Stream Example

Dispatch D continues to keep the remaining 8 cores busy
Dual Command Stream Example

DISPATCH(B, 8)

FLUSH

DISPATCH(D, 80)

48

1200 cy
Dual Command Stream Example

DISPATCH(B, 8)

FLUSH

DISPATCH(D, 80)

1600 cy
Did Two Front-Ends Help?

- It sure did!
  - ~98% utilization!
  - No additional cores
- Lower total execution time for $A + B + C + D$
- Higher latency for $A+B$ or $C+D$ submitted individually
Even Better For Real GPUs!

- Threads stalled on memory access
  - Real GPU’s will cycle threads on cores
- Idle time from cache flushes
- Tasks with limited shader core usage
  - Depth-only rasterization
  - On-Chip Tessellation/GS
  - DMA
Thinking in CPU Terms

- Multiple front-ends ≈ SMT
  - Simultaneous Multithreading (Hyperthreading)
- Interleave two instruction streams that share execution resources
- Similar goal: reduce idle time from stalls
Real-World Example: Bloom + DOF

Independent command streams
Real-World Example: Bloom + DOF
Submitting Commands in D3D12

- App records + submits command list(s)
  - With fences for synchronization
- OS schedules commands to run on an **engine**
  - Engine = driver exposed HW queue
  - Direct, compute, copy, and video
- HW command processor executes commands
Bloom + DOF in D3D12
Bloom + DOF in D3D12
D3D12 Multi-Queue Submission

- Submissions to multiple command queues will possibly execute concurrently
  - Depends on the OS scheduler
  - Depends on the GPU
  - Depends on the driver
  - Depends on the queue/command list type
  - Similar to threads on a CPU
D3D12 Virtualizes Queues

- D3D12 command queues ≠ hardware queues
- Hardware may have many queues, or only 1!
- The OS/scheduler will figure it out for you
  - Flattening of parallel submissions
  - Dependencies visible to scheduler via fences
- Check GPUView/PIX/RGP/Nsight to see what’s going on!
Vulkan Queues Are Different!

- They’re not virtualized!
  - ...or at least not in the same way
- Query at runtime for “queue families”
  - Vk queue family $\approx$ D3D12 engine
- Explicit bind to exposed queue
  - Still not guaranteed to be a HW queue
Using Async Compute

- Fills in idle shader cores
  - Just like our MJP-4000 example!
- Identify independent command streams
  - ...and submit them on separate queues
- Works best when lots of cores are idle
  - Depth-only rendering
  - Lots of barriers
Recap
GPU Barriers Ensure Data Visibility

- Probably involves GPU thread sync
- Maybe involves cache flushes
- Maybe involves data transformation
  - Decompression
- API barriers describe visibility + dependencies
  - Think about your dependencies! (or visualize them!)
GPUs Aren’t *That* Different

- Command processor = task scheduler
- Shader cores = worker cores
- Multi-core CPU’s have similar problems!
  - Parallel operations
  - Coherency issues
Barriers = Idle Cores

- Keep the thread monster fed!
  - Waits/stalls decrease utilization
  - Careful barrier use => higher utilization
  - Watch out for long-running threads!
- Batch your barriers!
  - Flushing cache once >>> flushing multiple times
Using Multiple Queues

- Parallel submissions *may* increase utilization
  - Not guaranteed! – check your tools!
- Won’t magically increase the core count
- Look for independent command streams
  - Don’t go crazy with D3D12 fences
That’s It!

- Thanks to...
  - Ste Tovey
  - Rys Sommefeldt
  - Nick Thibieroz
  - Andrei Tatarinov
  - Everyone at Ready At Dawn
Contact Info

- matt@readyatdawn.com
- mpettineo@gmail.com
- @mynameismjp
- https://mynameismjp.wordpress.com/
- https://github.com/TheRealMJP