GDC

Breaking Down Barriers: An Intro to GPU Synchronization

Matt Pettineo
Lead Engine Programmer
Ready At Dawn Studios

GAME DEVELOPERS CONFERENCE
MARCH 18–22, 2019 | #GDC19

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!

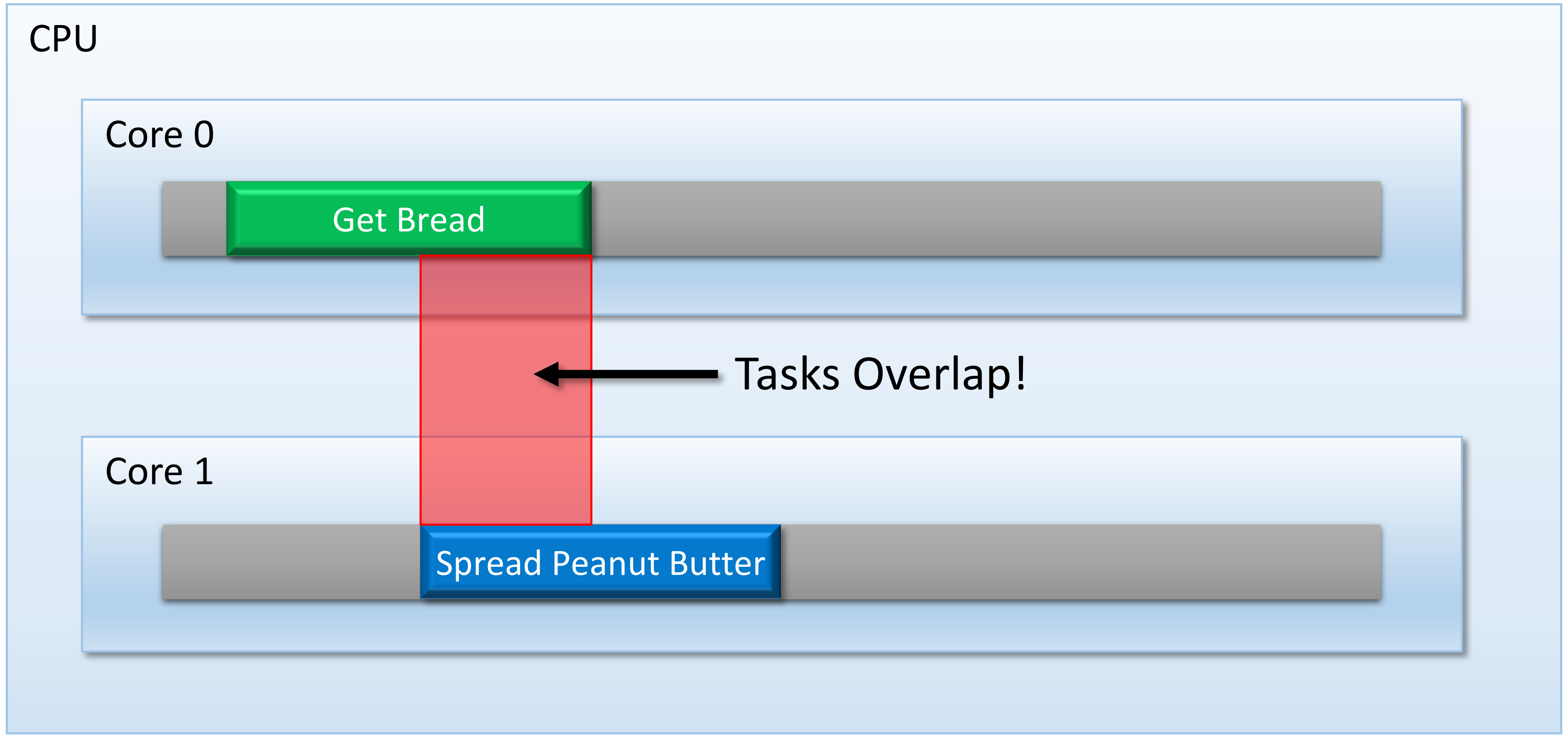
Single-Threaded Dependencies

- 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

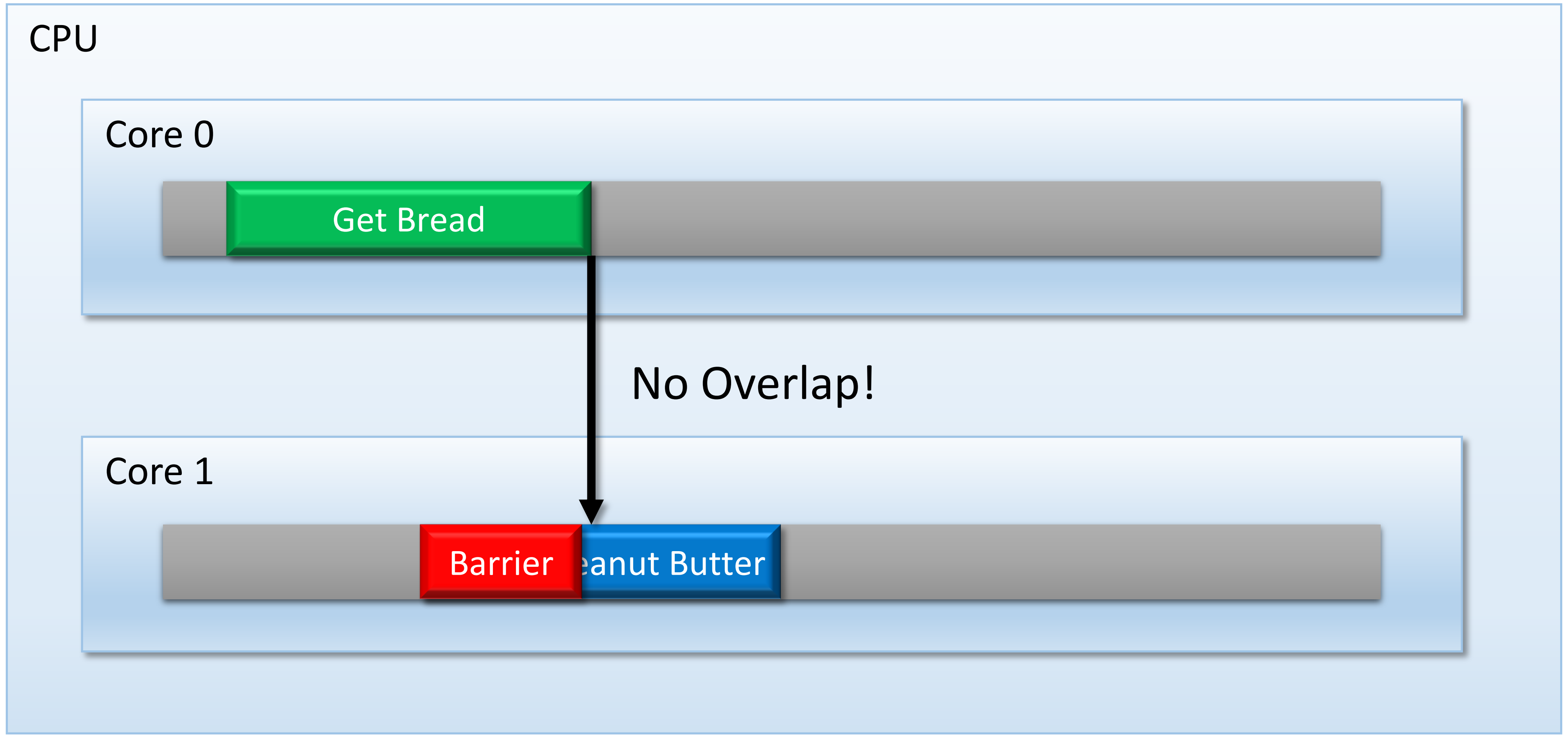
Multi-Threaded Dependencies

- 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

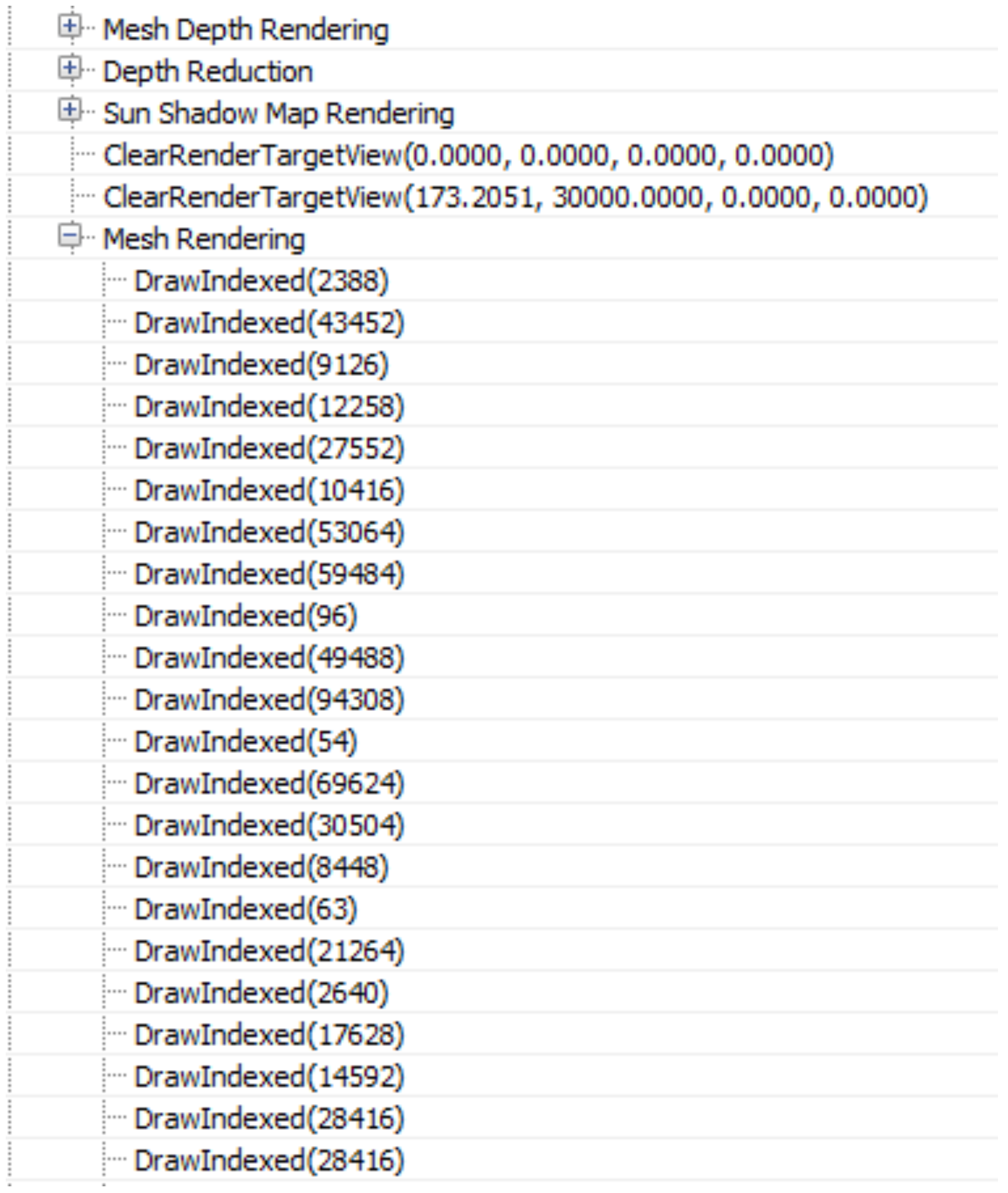


Task Dependencies

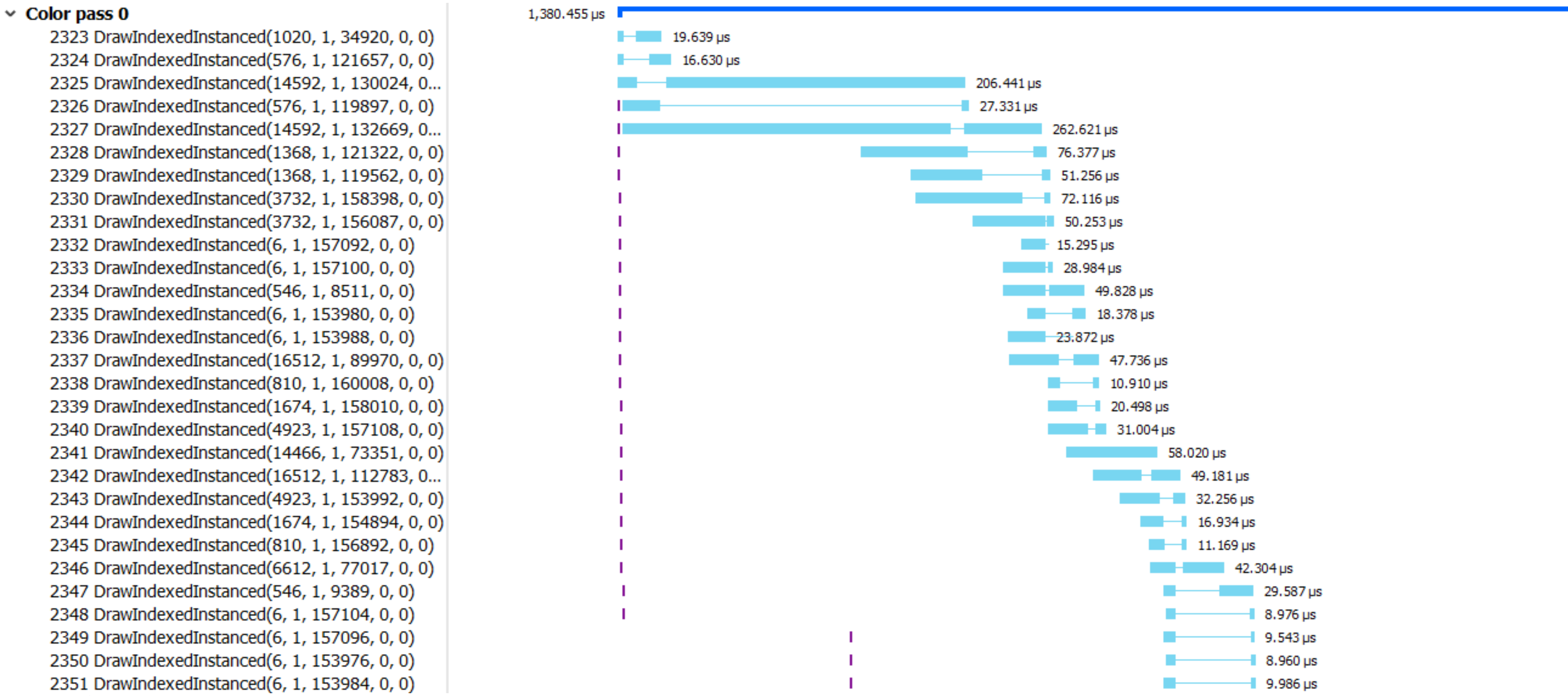


GPU Parallelism

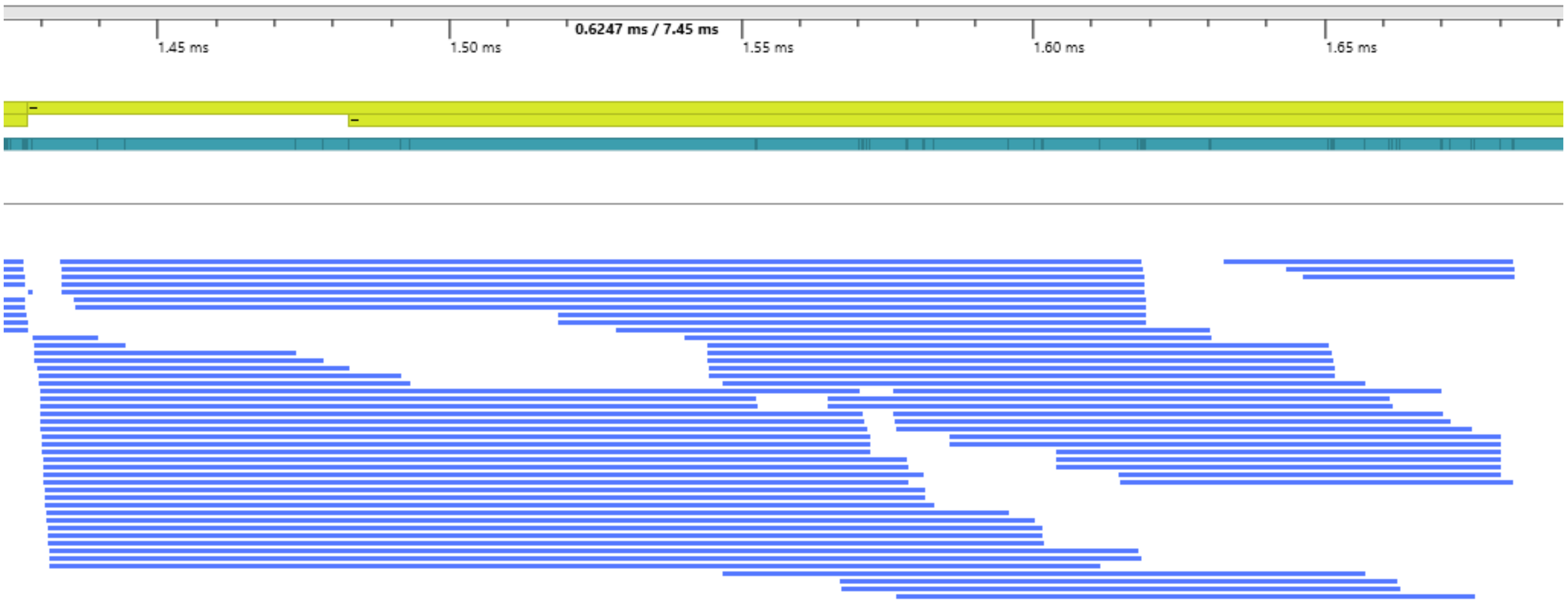
- GPU is **<u>not</u>** a serial machine!
  - Looks are deceiving
  - HW and drivers help you out



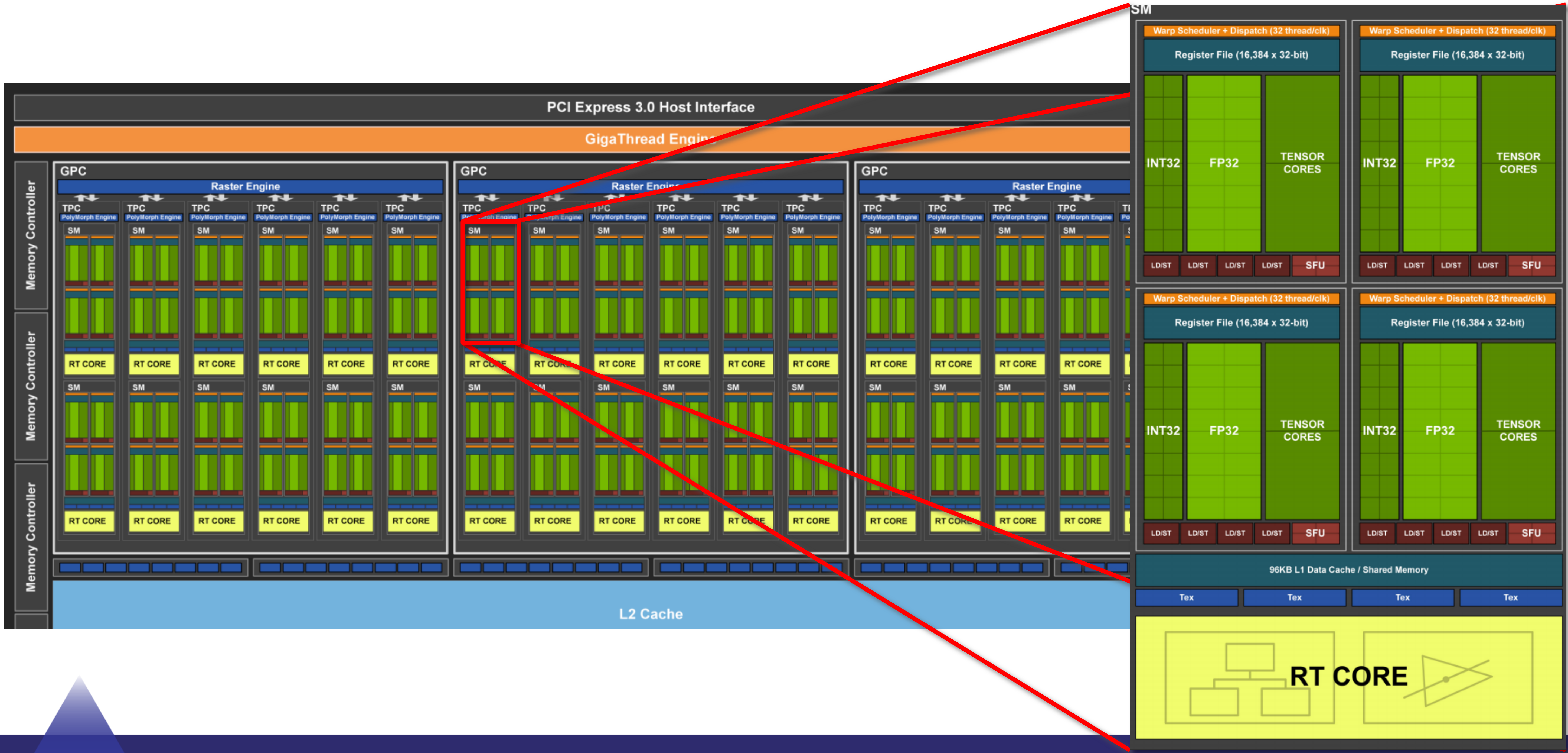
```
⊞ Mesh Depth Rendering
⊞ Depth Reduction
⊞ Sun Shadow Map Rendering
  ClearRenderTargetView(0.0000, 0.0000, 0.0000, 0.0000)
  ClearRenderTargetView(173.2051, 30000.0000, 0.0000, 0.0000)
⊟ Mesh Rendering
    DrawIndexed(2388)
    DrawIndexed(43452)
    DrawIndexed(9126)
    DrawIndexed(12258)
    DrawIndexed(27552)
    DrawIndexed(10416)
    DrawIndexed(53064)
    DrawIndexed(59484)
    DrawIndexed(96)
    DrawIndexed(49488)
    DrawIndexed(94308)
    DrawIndexed(54)
    DrawIndexed(69624)
    DrawIndexed(30504)
    DrawIndexed(8448)
    DrawIndexed(63)
    DrawIndexed(21264)
    DrawIndexed(2640)
    DrawIndexed(17628)
    DrawIndexed(14592)
    DrawIndexed(28416)
    DrawIndexed(28416)
```



```
Color pass 0                                          1,380.455 µs
  2323 DrawIndexedInstanced(1020, 1, 34920, 0, 0)     19.639 µs
  2324 DrawIndexedInstanced(576, 1, 121657, 0, 0)     16.630 µs
  2325 DrawIndexedInstanced(14592, 1, 130024, 0...    206.441 µs
  2326 DrawIndexedInstanced(576, 1, 119897, 0, 0)     27.331 µs
  2327 DrawIndexedInstanced(14592, 1, 132669, 0...    262.621 µs
  2328 DrawIndexedInstanced(1368, 1, 121322, 0, 0)    76.377 µs
  2329 DrawIndexedInstanced(1368, 1, 119562, 0, 0)    51.256 µs
  2330 DrawIndexedInstanced(3732, 1, 158398, 0, 0)    72.116 µs
  2331 DrawIndexedInstanced(3732, 1, 156087, 0, 0)    50.253 µs
  2332 DrawIndexedInstanced(6, 1, 157092, 0, 0)       15.295 µs
  2333 DrawIndexedInstanced(6, 1, 157100, 0, 0)       28.984 µs
  2334 DrawIndexedInstanced(546, 1, 8511, 0, 0)       49.828 µs
  2335 DrawIndexedInstanced(6, 1, 153980, 0, 0)       18.378 µs
  2336 DrawIndexedInstanced(6, 1, 153988, 0, 0)       23.872 µs
  2337 DrawIndexedInstanced(16512, 1, 89970, 0, 0)    47.736 µs
  2338 DrawIndexedInstanced(810, 1, 160008, 0, 0)     10.910 µs
  2339 DrawIndexedInstanced(1674, 1, 158010, 0, 0)    20.498 µs
  2340 DrawIndexedInstanced(4923, 1, 157108, 0, 0)    31.004 µs
  2341 DrawIndexedInstanced(14466, 1, 73351, 0, 0)    58.020 µs
  2342 DrawIndexedInstanced(16512, 1, 112783, 0...    49.181 µs
  2343 DrawIndexedInstanced(4923, 1, 153992, 0, 0)    32.256 µs
  2344 DrawIndexedInstanced(1674, 1, 154894, 0, 0)    16.934 µs
  2345 DrawIndexedInstanced(810, 1, 156892, 0, 0)     11.169 µs
  2346 DrawIndexedInstanced(6612, 1, 77017, 0, 0)     42.304 µs
  2347 DrawIndexedInstanced(546, 1, 9389, 0, 0)       29.587 µs
  2348 DrawIndexedInstanced(6, 1, 157104, 0, 0)       8.976 µs
  2349 DrawIndexedInstanced(6, 1, 157096, 0, 0)       9.543 µs
  2350 DrawIndexedInstanced(6, 1, 153976, 0, 0)       8.960 µs
  2351 DrawIndexedInstanced(6, 1, 153984, 0, 0)       9.986 µs
```


0.6247 ms / 7.45 ms
1.45 ms
1.50 ms
1.55 ms
1.60 ms
1.65 ms

GPUs are Thread Monsters!


PCI Express 3.0 Host Interface
GigaThread Engine
Memory Controller
GPC
Raster Engine
TPC
PolyMorph Engine
SM
RT CORE
L2 Cache
SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

GPUs are Thread Monsters!

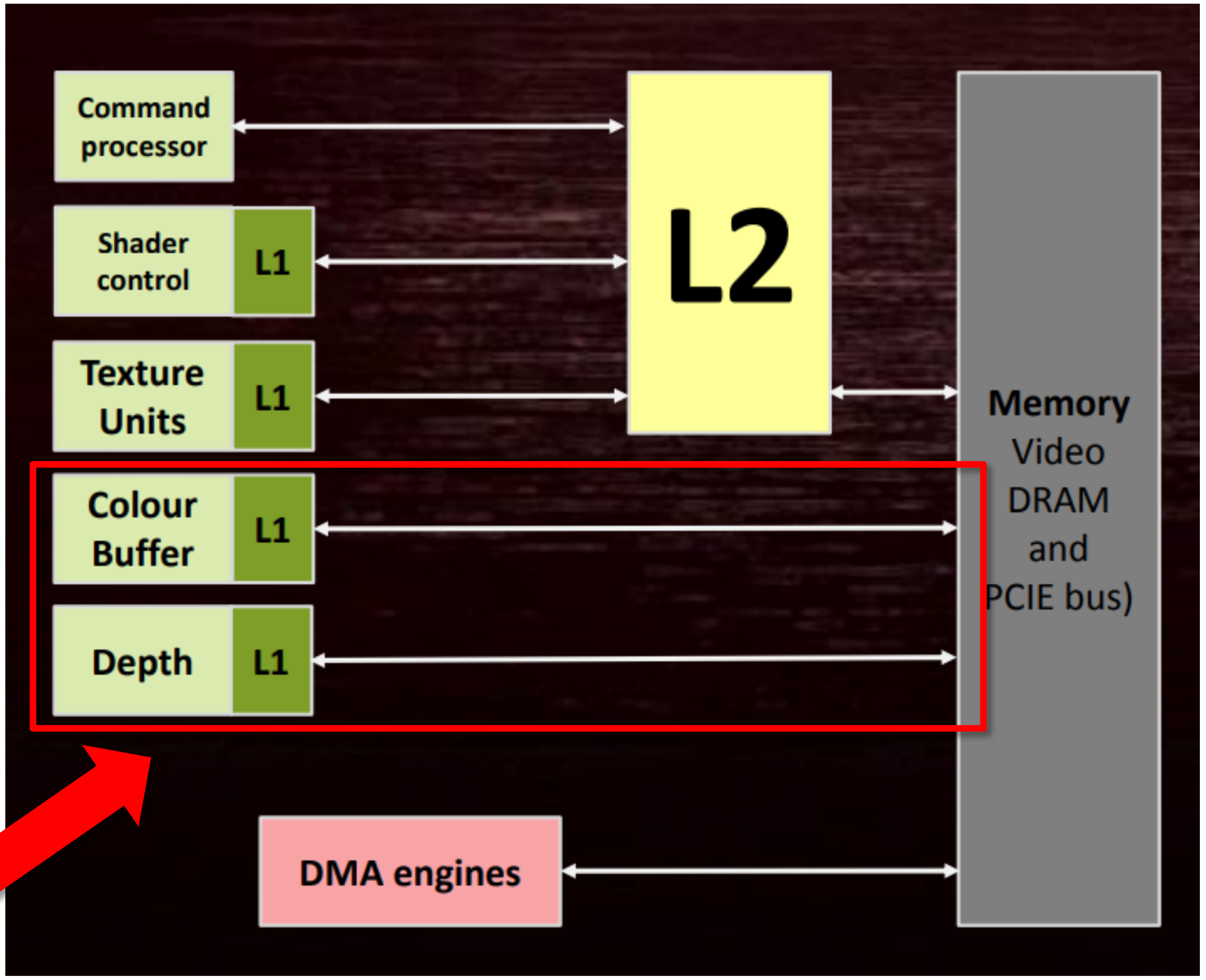
- Lots of overlapping when possible
  - No dependencies
  - Re-ordering for render target writes (ROPs)
- Overlap improves performance!
  - More on this later

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

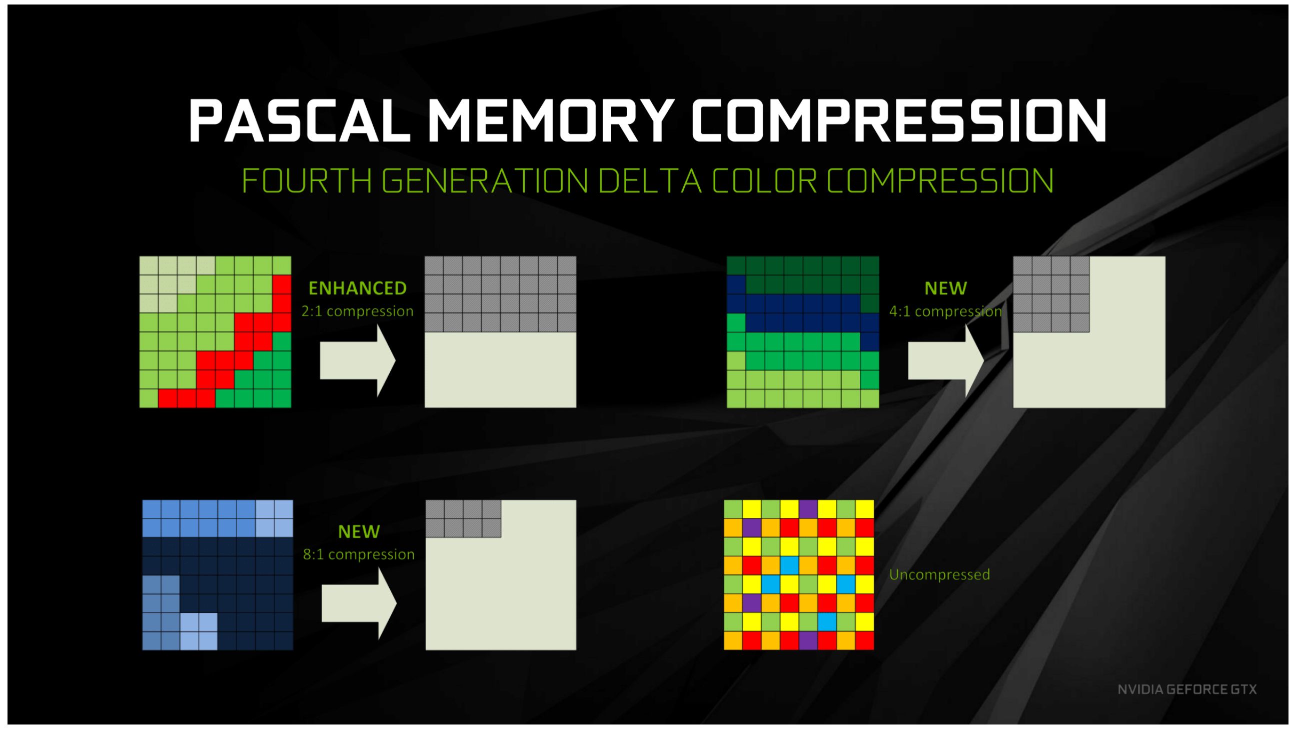
- Lots of caches!
- Not always coherent!
  - Different from CPU's
- Flush and/or invalidate to ensure **visibility**
- **Batch your barriers!**



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

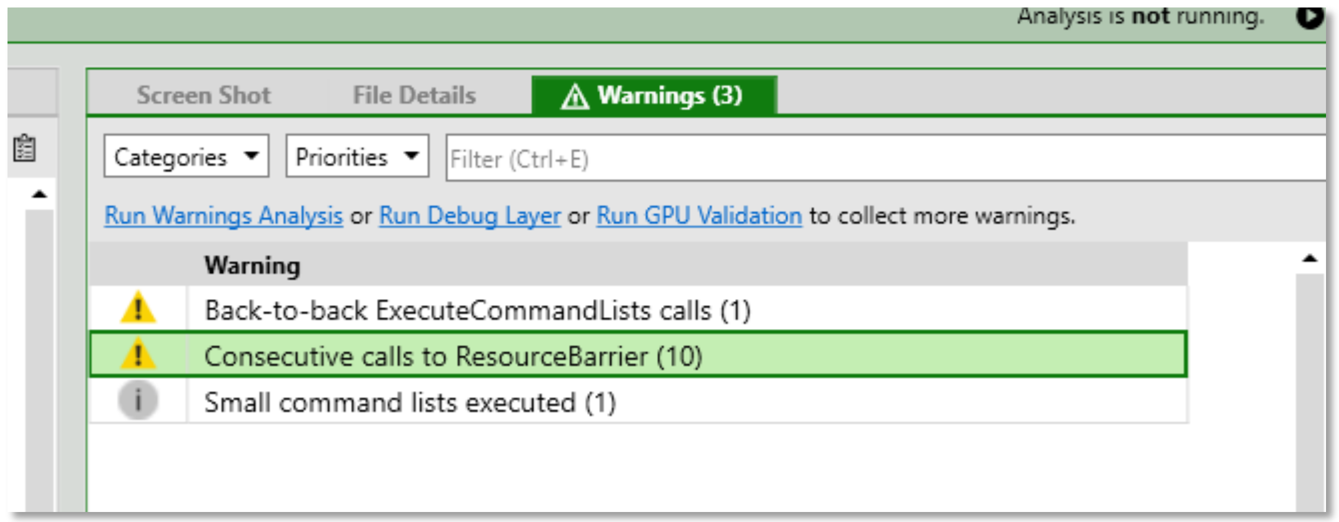
- Higher level - “resource state” abstraction
  - Texture is in an SRV read state
  - Buffer is in a UAV write state
  - Mostly describes resource **visibility**
- Implicit dependencies from state transition
- Layout/compression also implied

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!



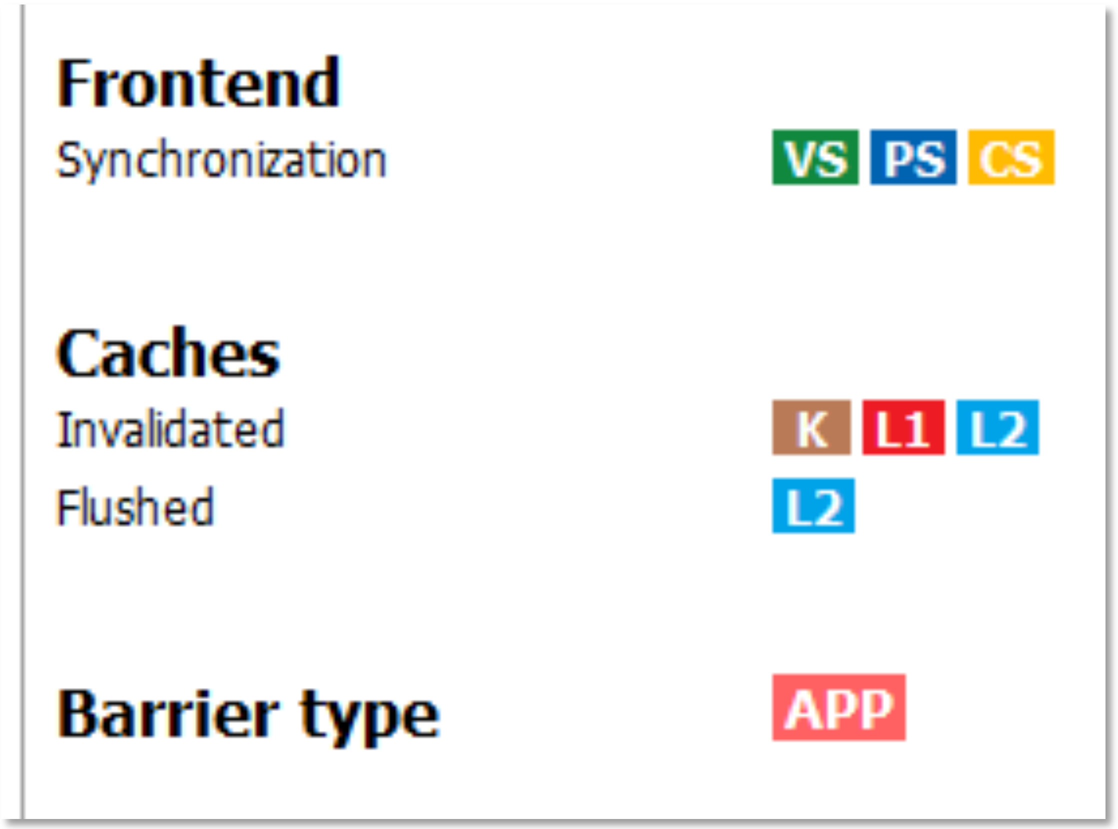
Analysis is **not** running.

Screen Shot | File Details | Warnings (3)

Categories | Priorities | Filter (Ctrl+E)

Run Warnings Analysis or Run Debug Layer or Run GPU Validation to collect more warnings.

	Warning
⚠	Back-to-back ExecuteCommandLists calls (1)
⚠	Consecutive calls to ResourceBarrier (10)
i	Small command lists executed (1)



Frontend

Synchronization	VS PS CS

Caches

Invalidated	K L1 L2
Flushed	L2

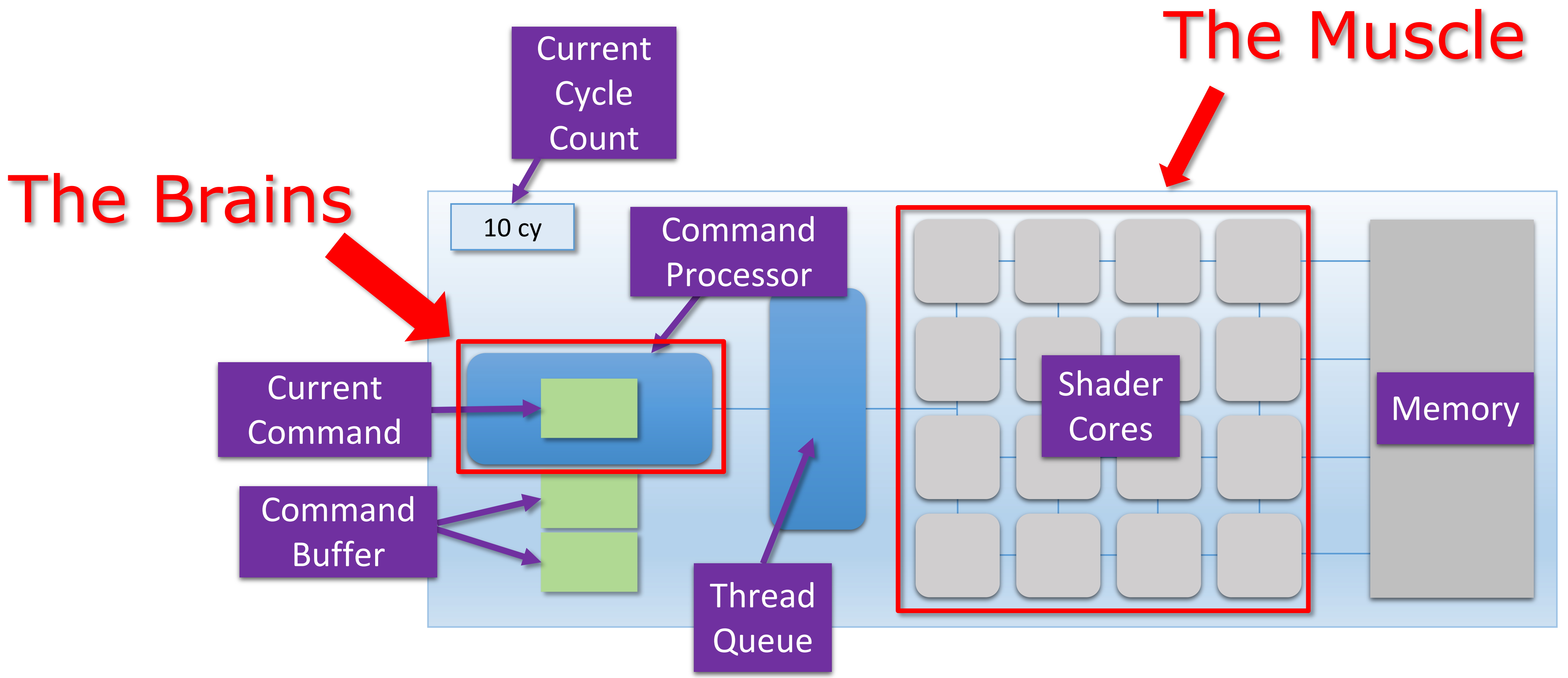
Barrier type	APP

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

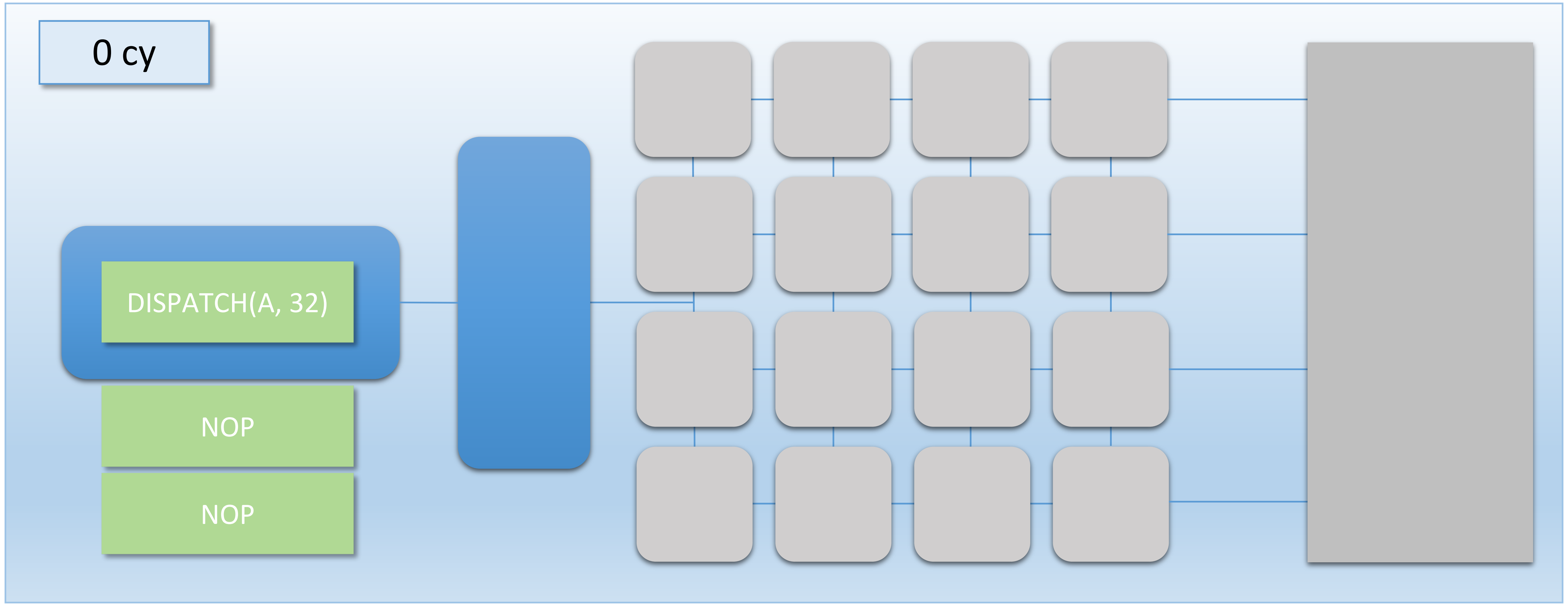
MJP-3000 Limitations

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

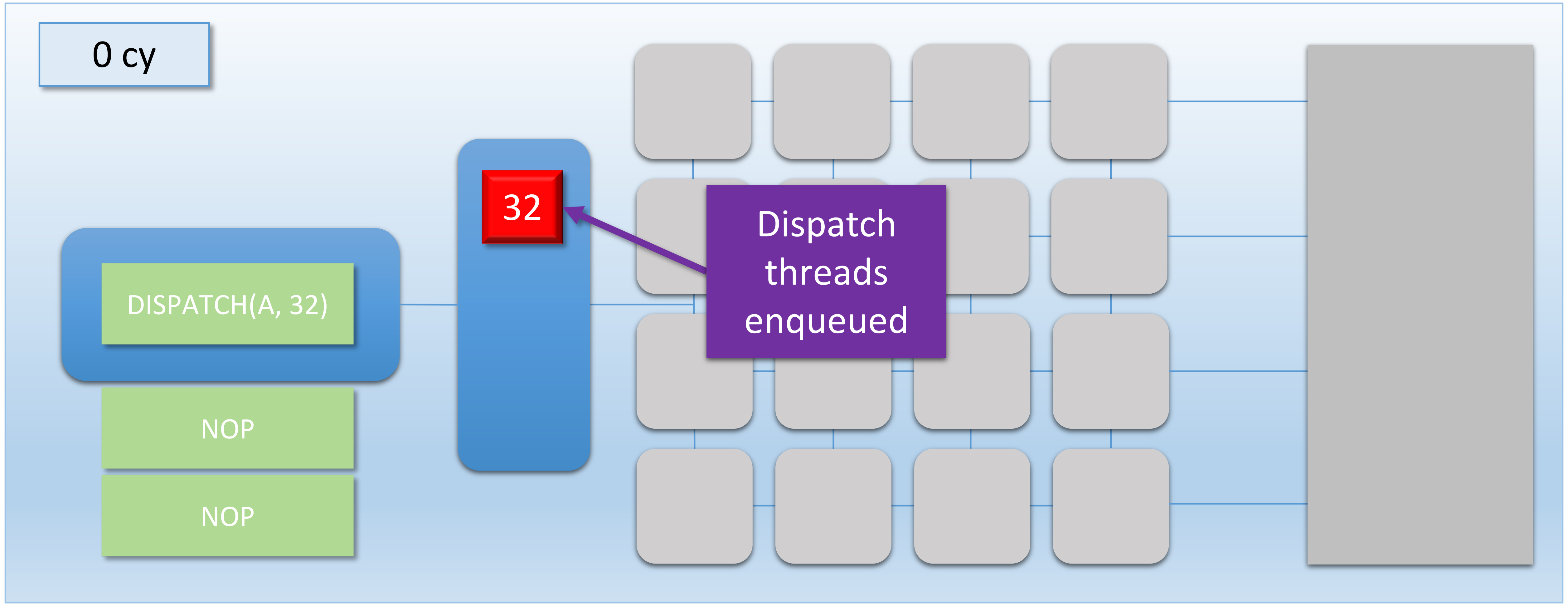
Simple Dispatch Example

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

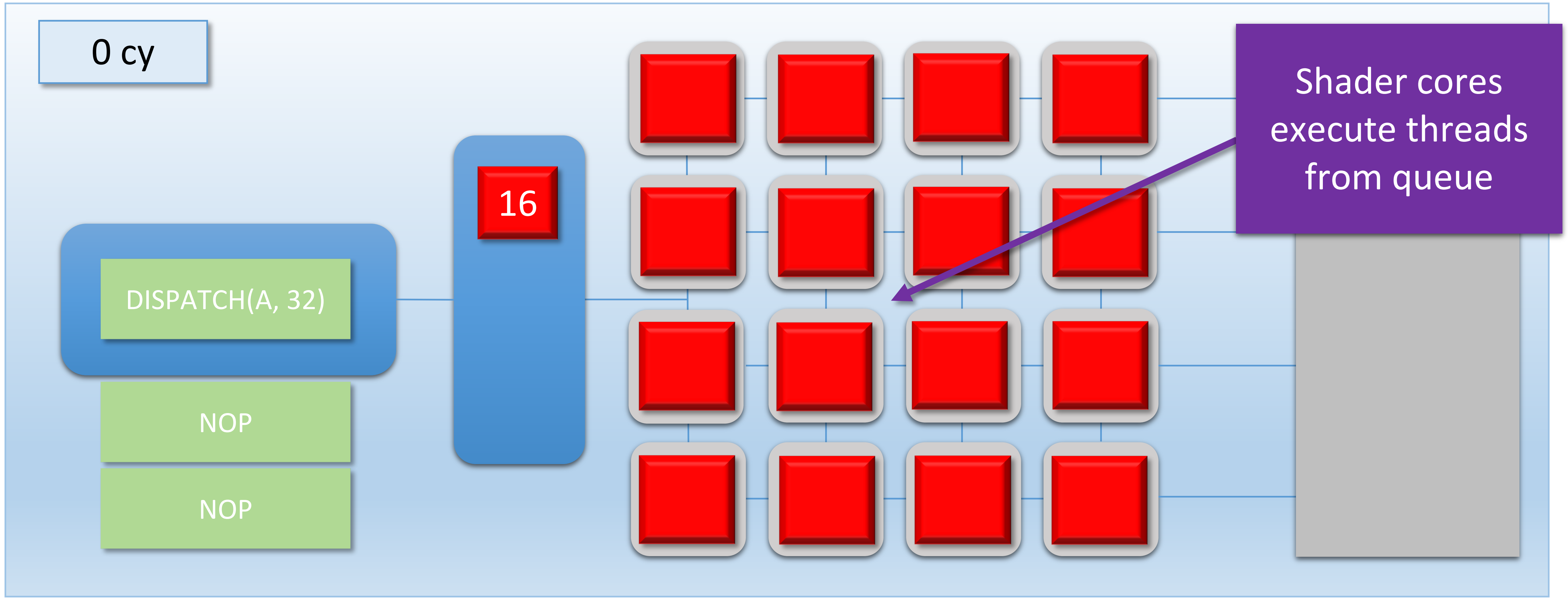
Simple Dispatch Example



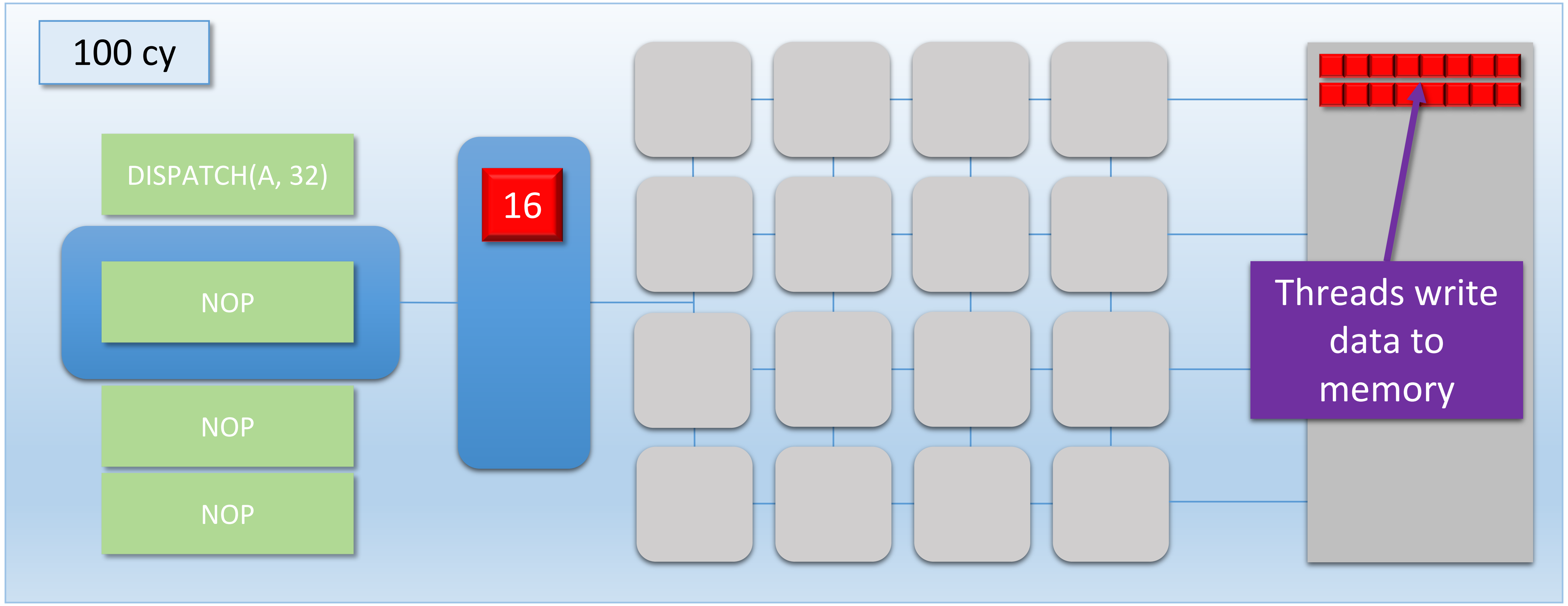
Simple Dispatch Example



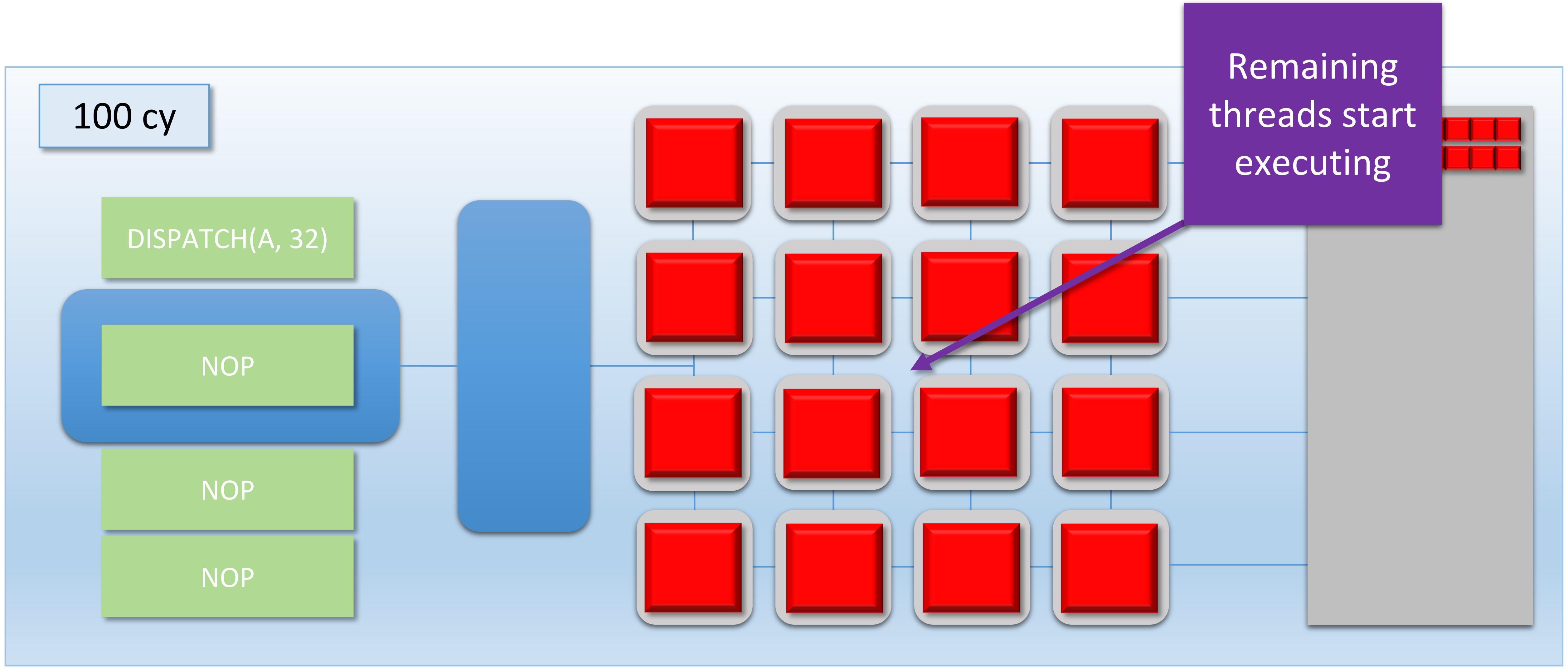
Simple Dispatch Example



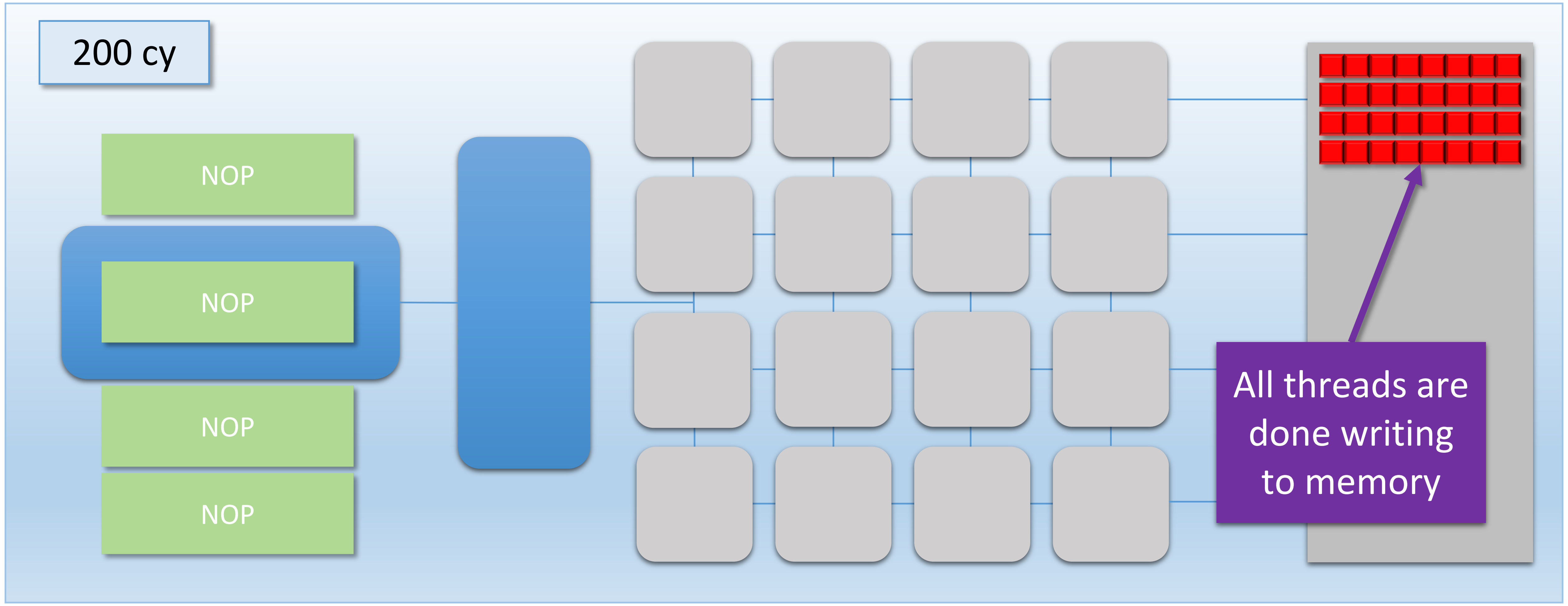
Simple Dispatch Example



Simple Dispatch Example



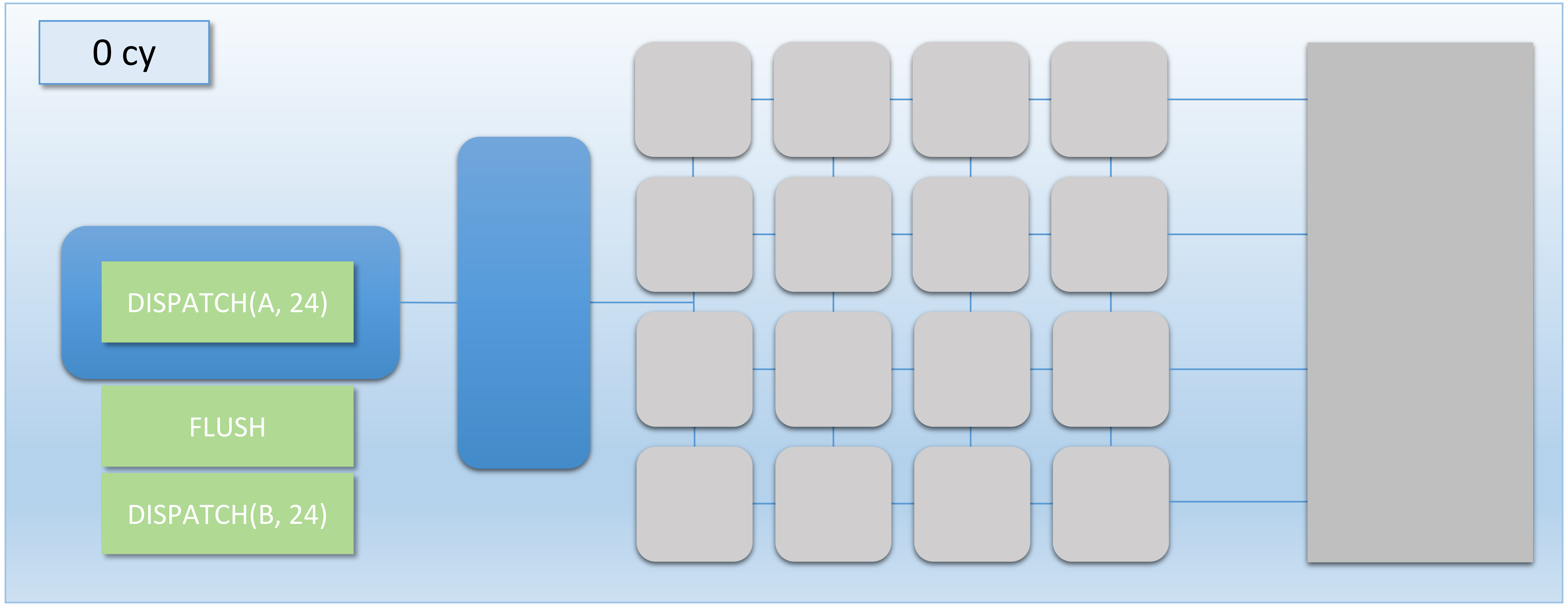
Simple Dispatch Example



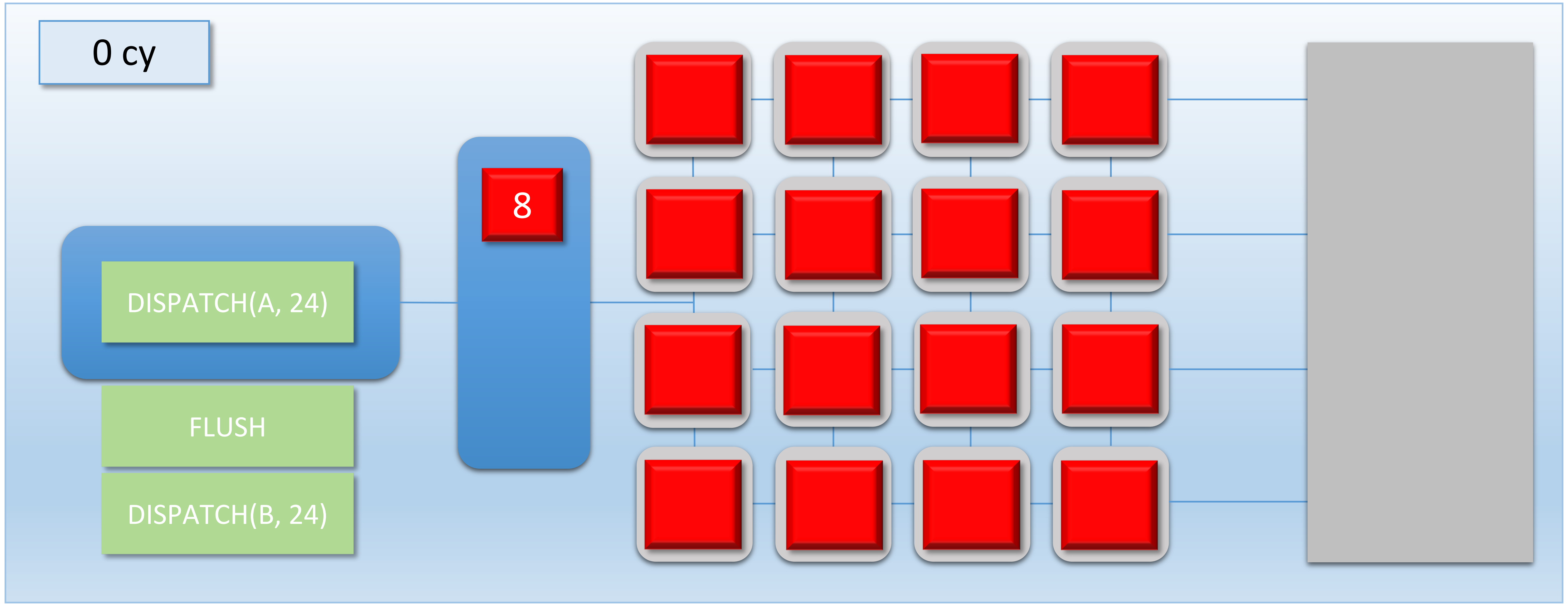
Thread Barrier Example

- 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

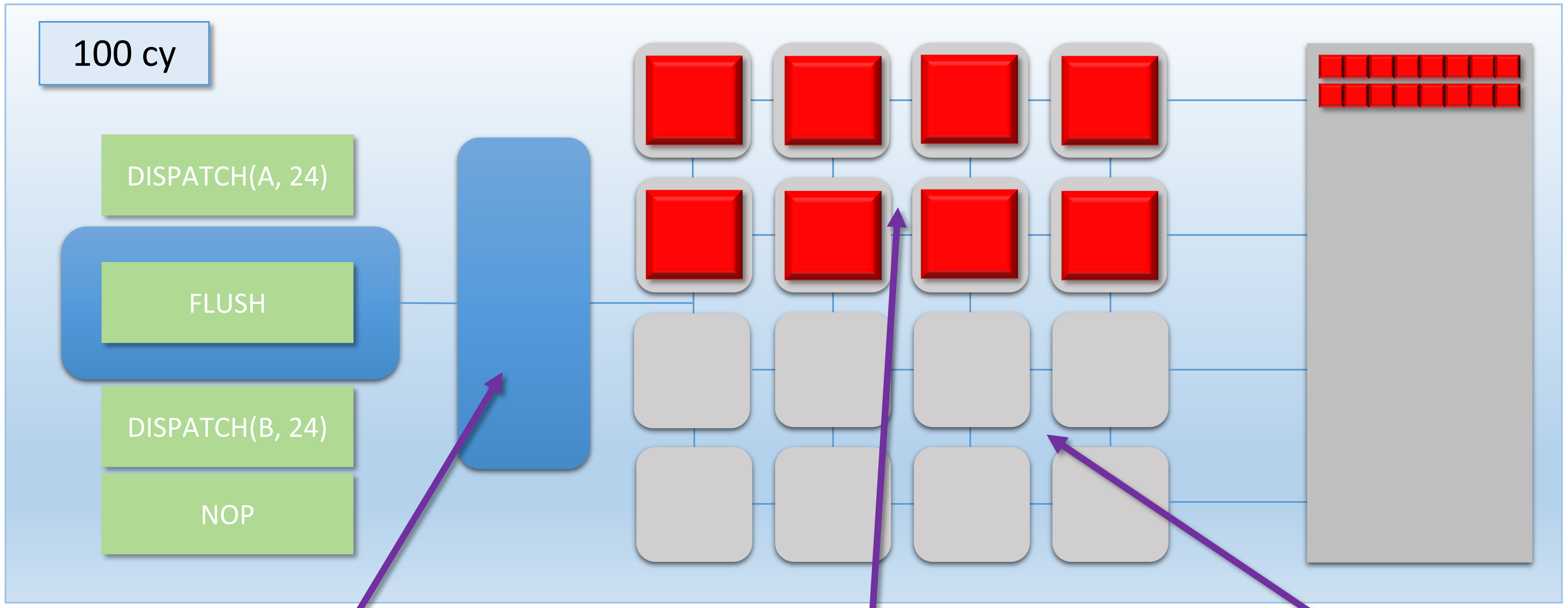
Thread Barrier Example



Thread Barrier Example



Thread Barrier Example

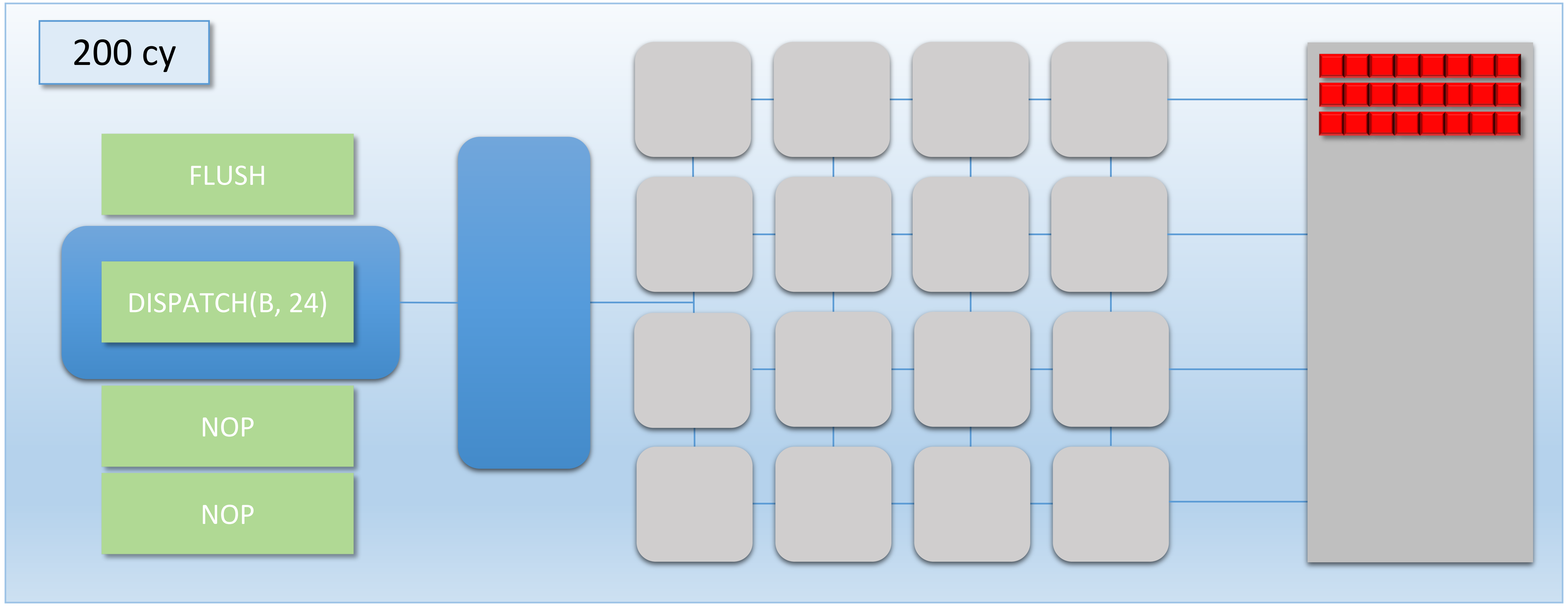


FLUSH waits for queue to empty

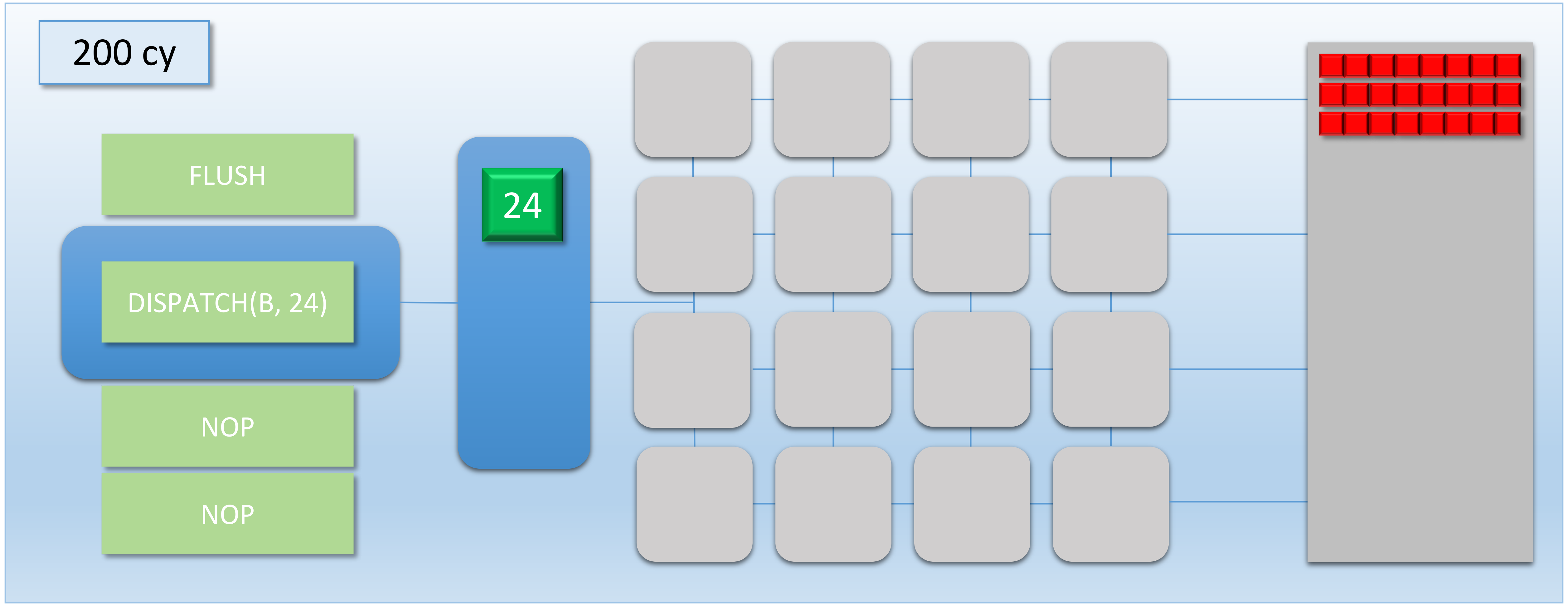
No overlap!

Cores are idle!

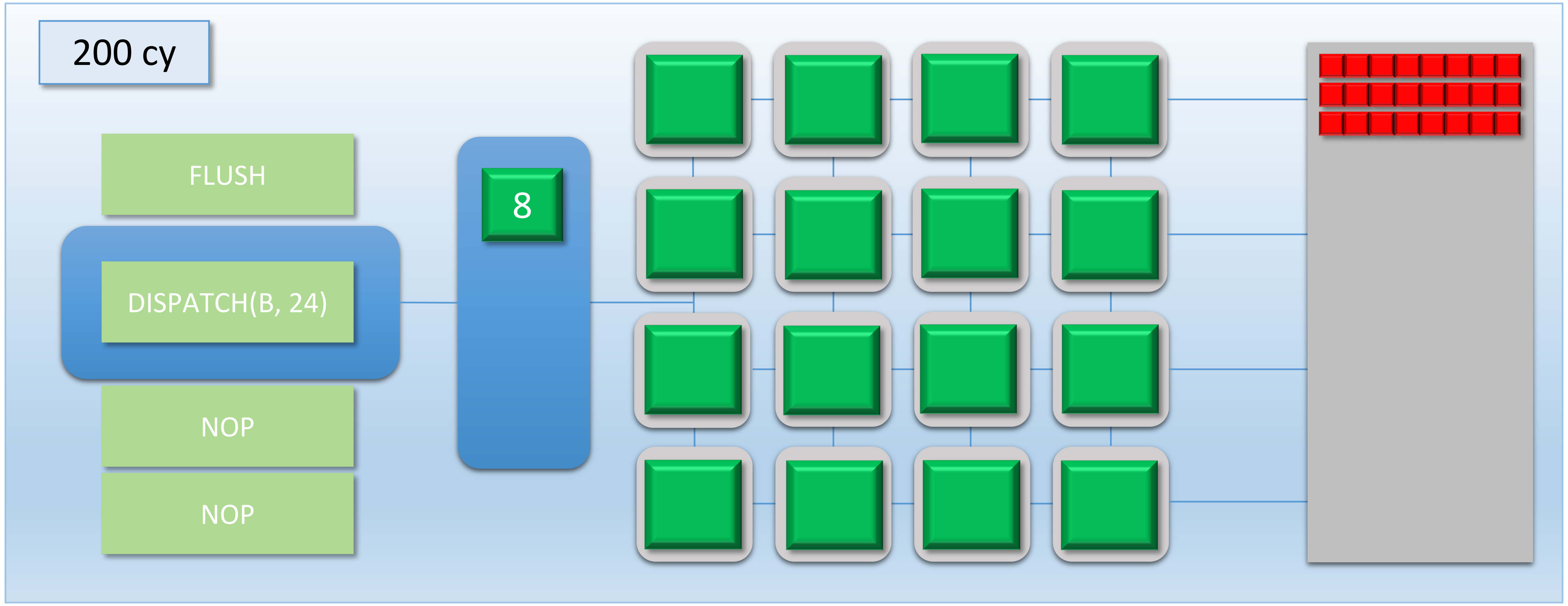
Thread Barrier Example



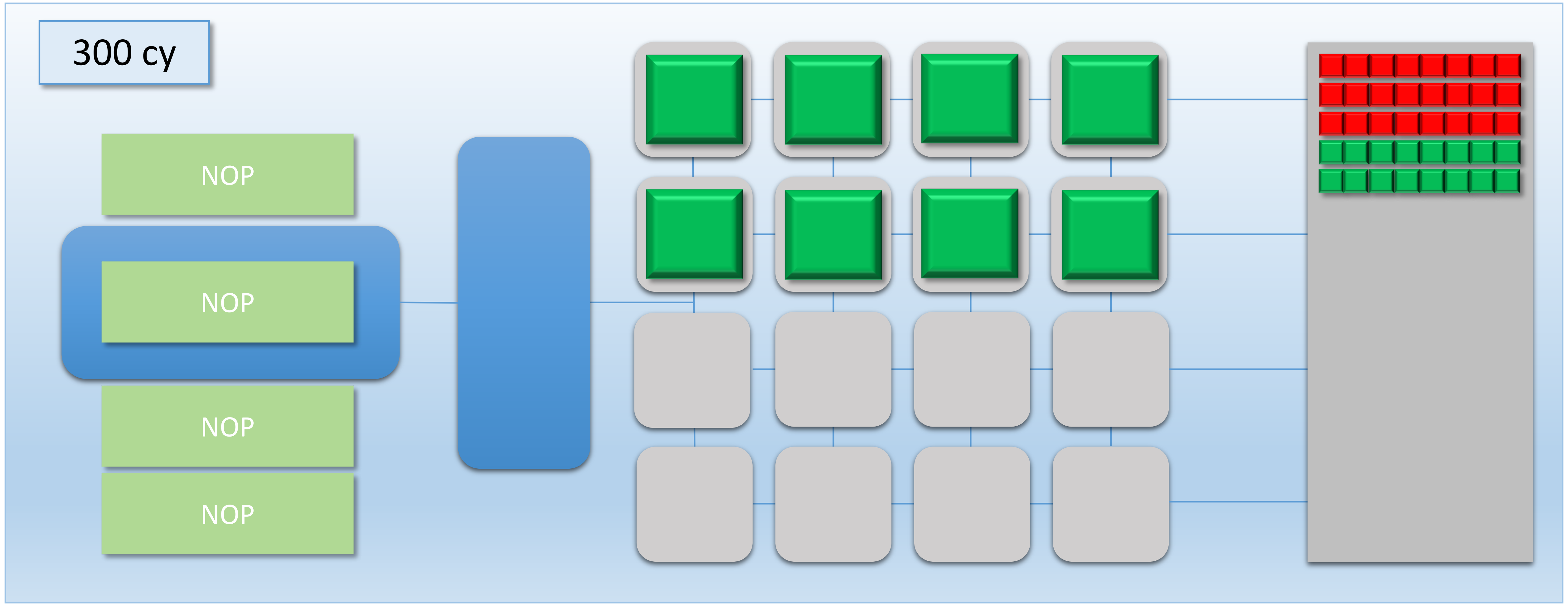
Thread Barrier Example



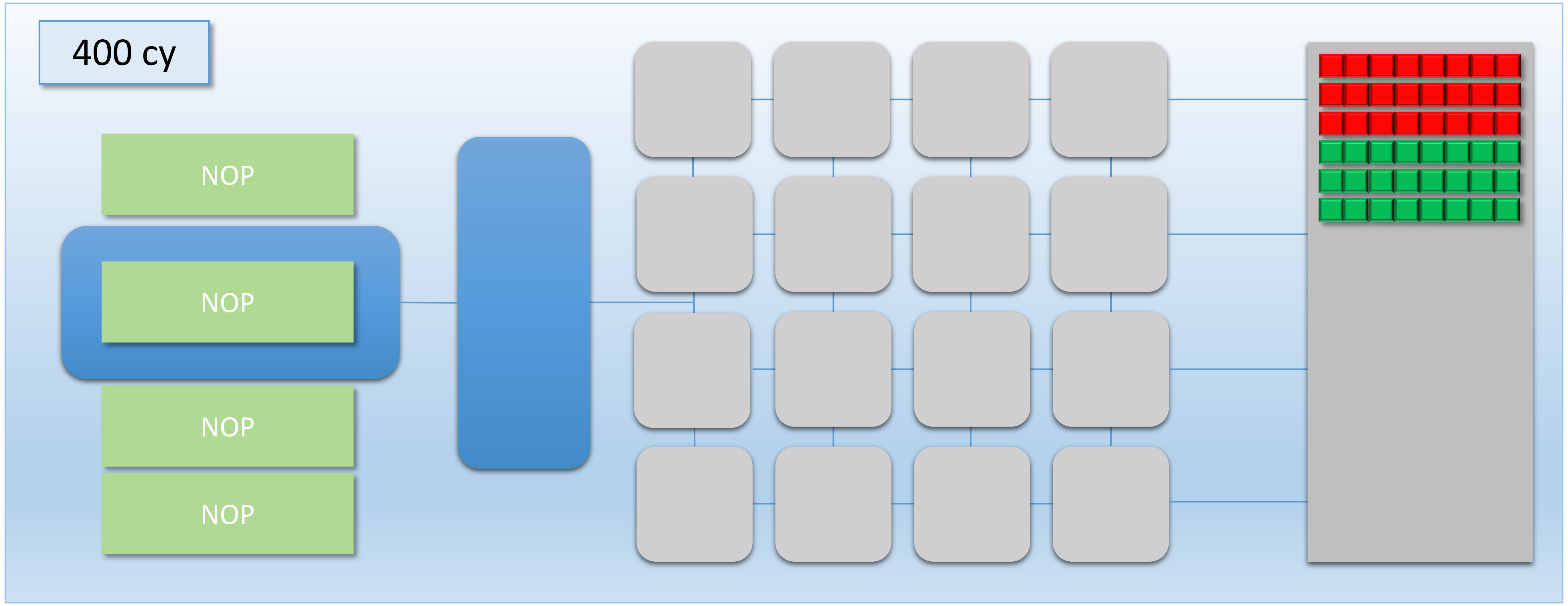
Thread Barrier Example



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

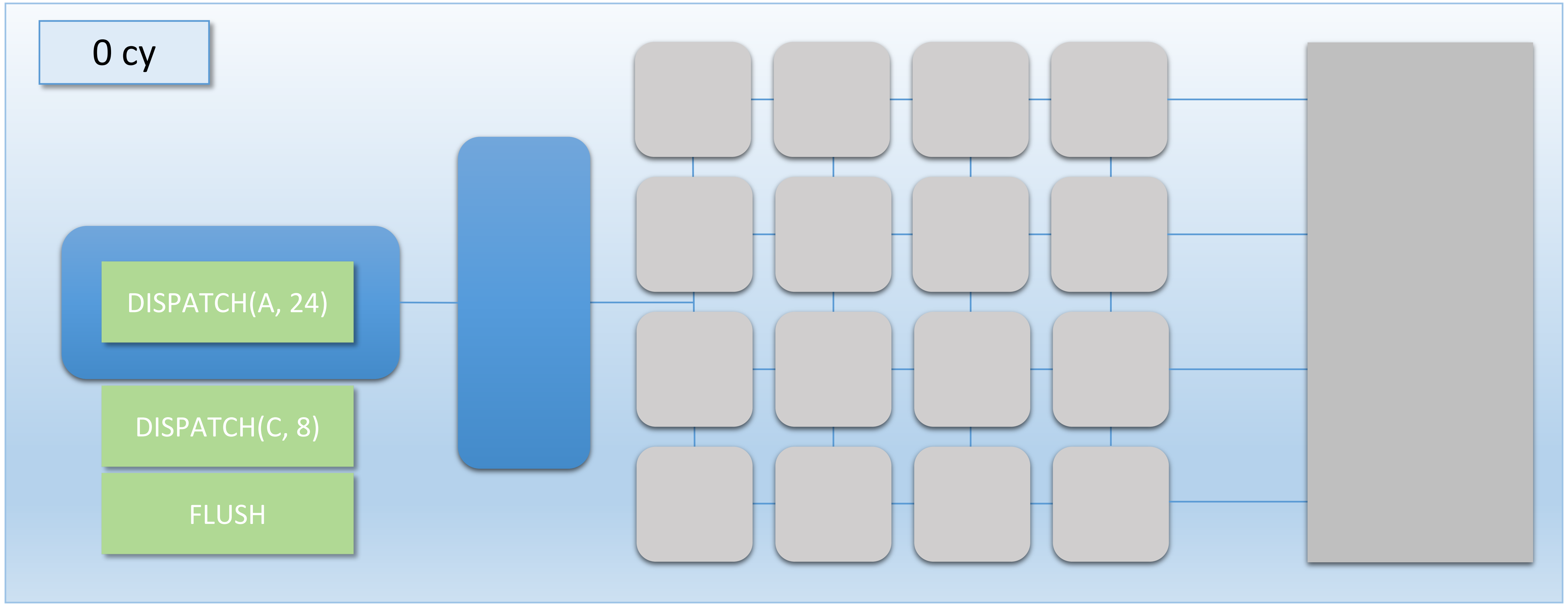
D3D12/Vulkan Barriers are Flushes!

- Expect a thread flush for a transition/pipeline barrier between draws/dispatches
- Same for a D3D12_RESOURCE_UAV_BARRIER
- Try to group non-dependent draws/dispatches between barriers
- May not be true for future GPUs!

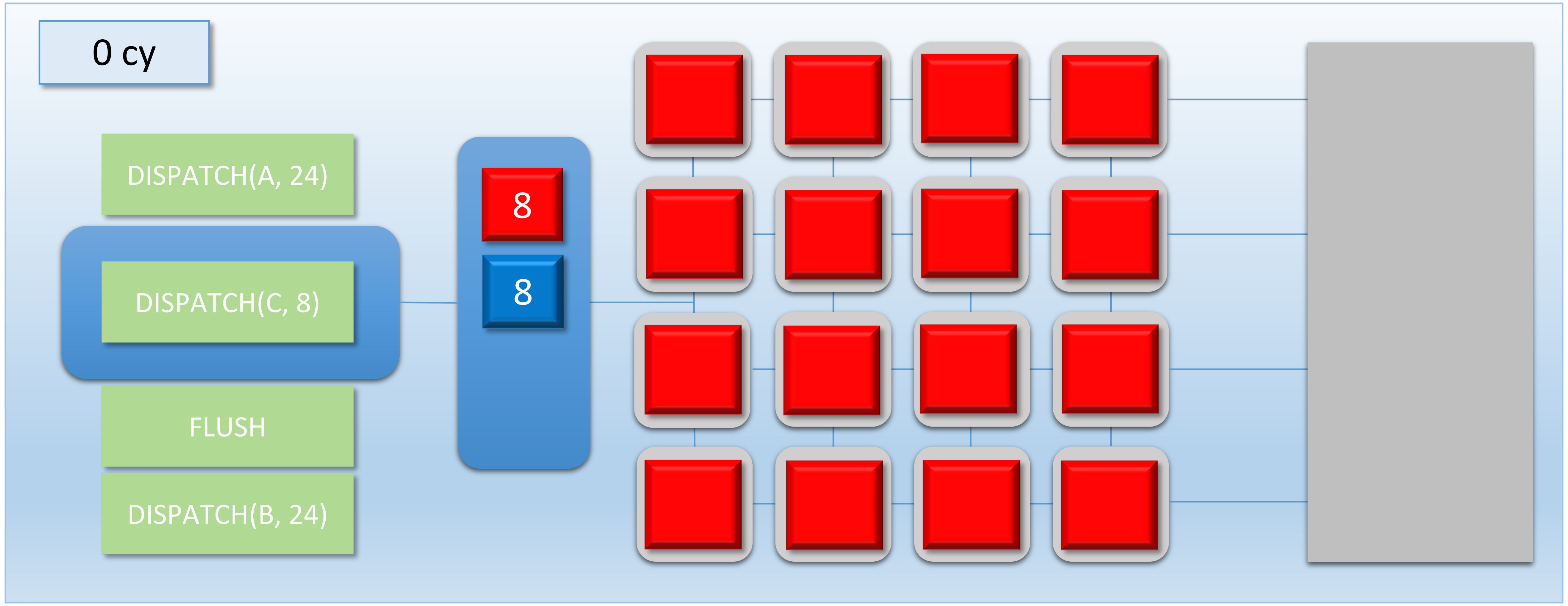
Overlapping Dispatches Example

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

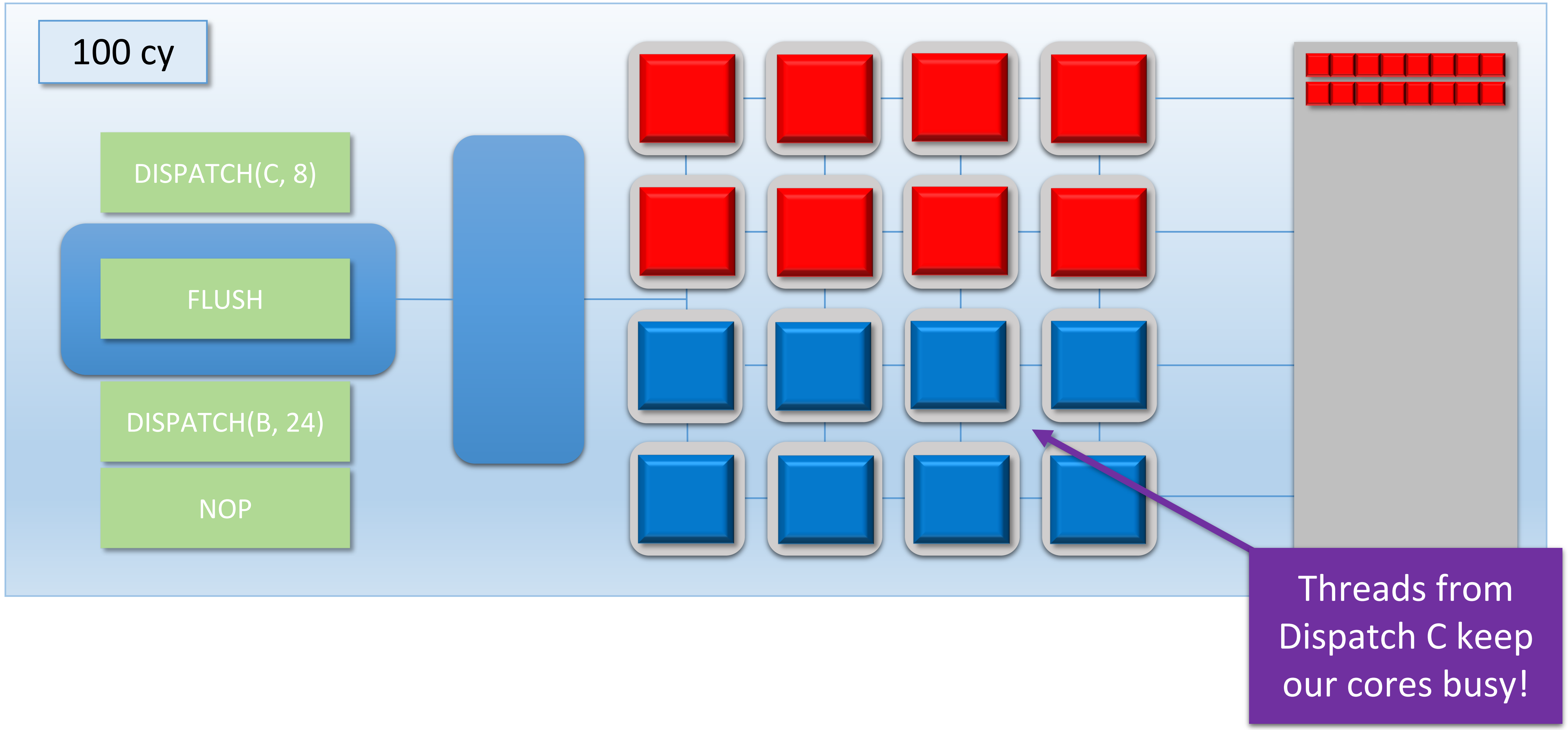
Overlapping Dispatches Example



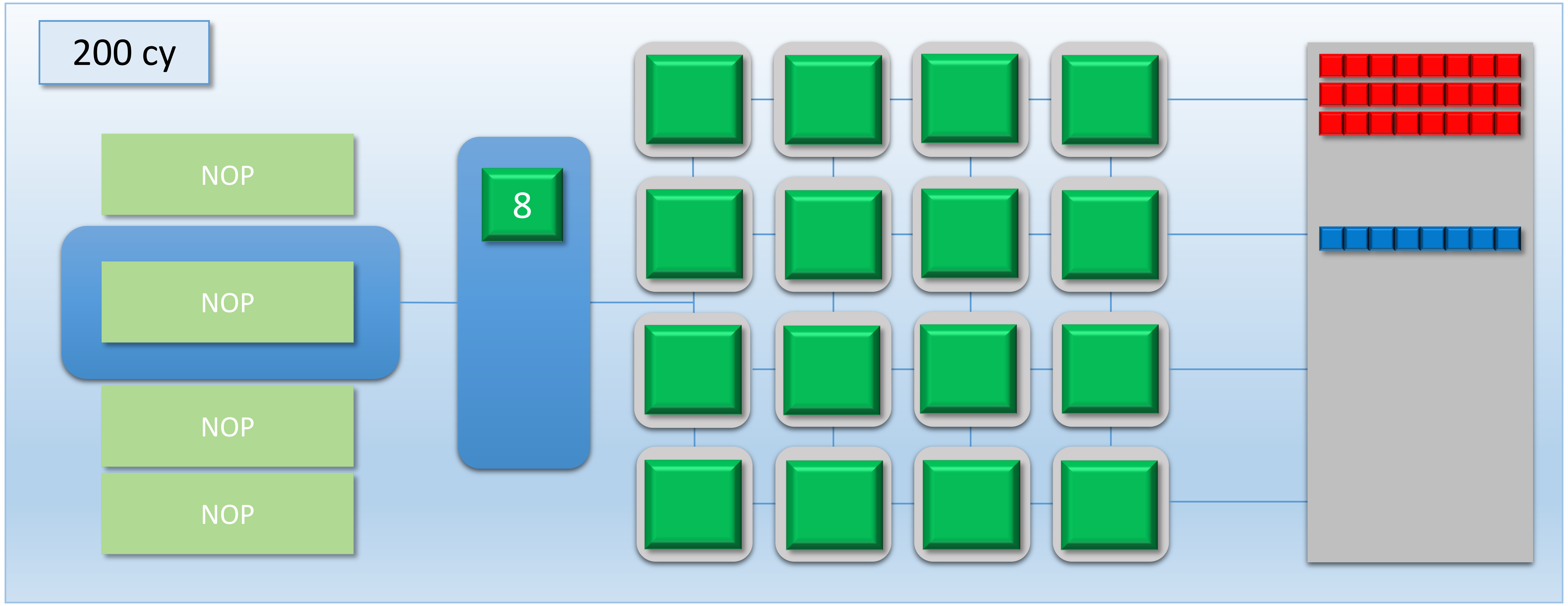
Overlapping Dispatches Example



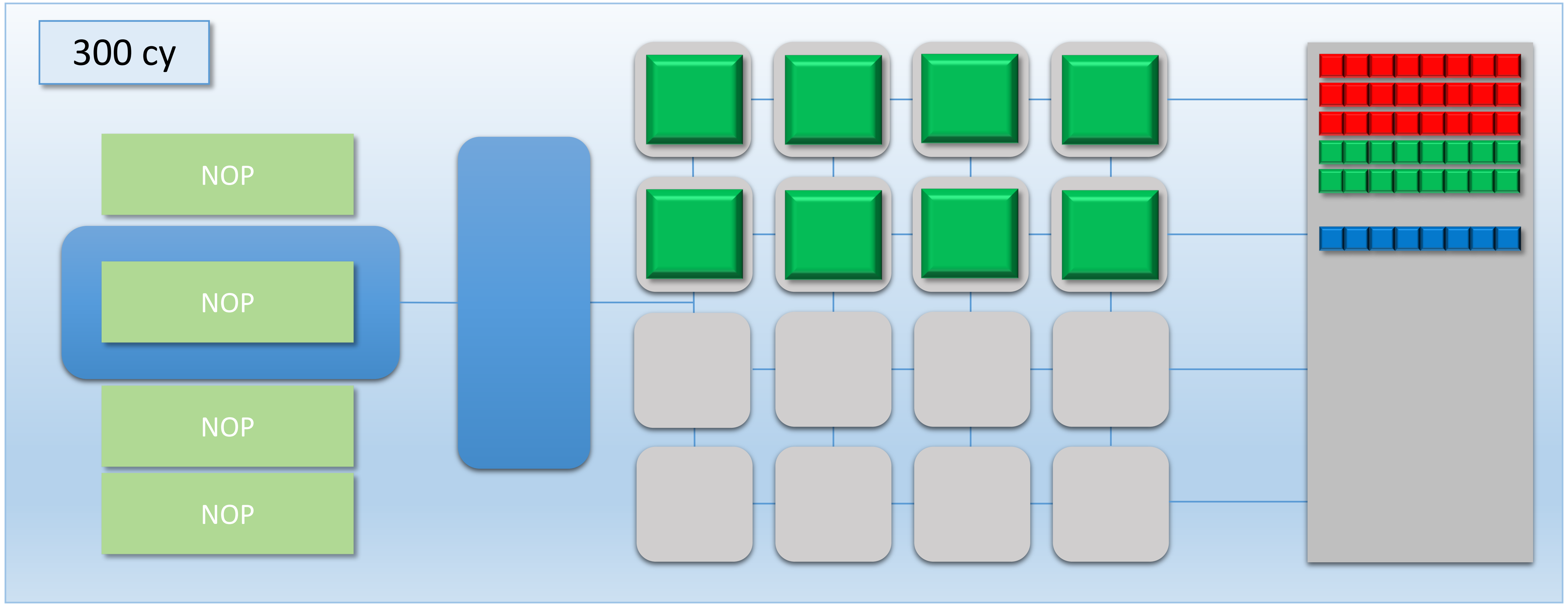
Overlapping Dispatches Example



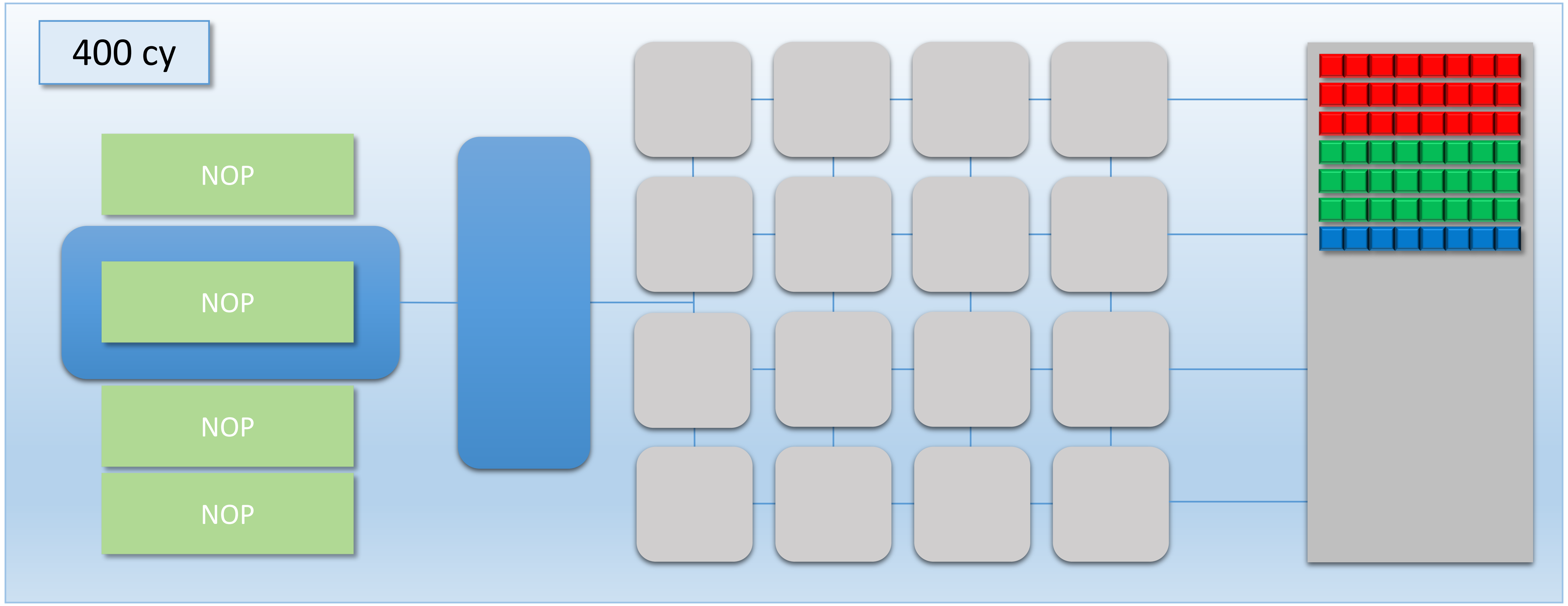
Overlapping Dispatches Example



Overlapping Dispatches Example



Overlapping Dispatches Example



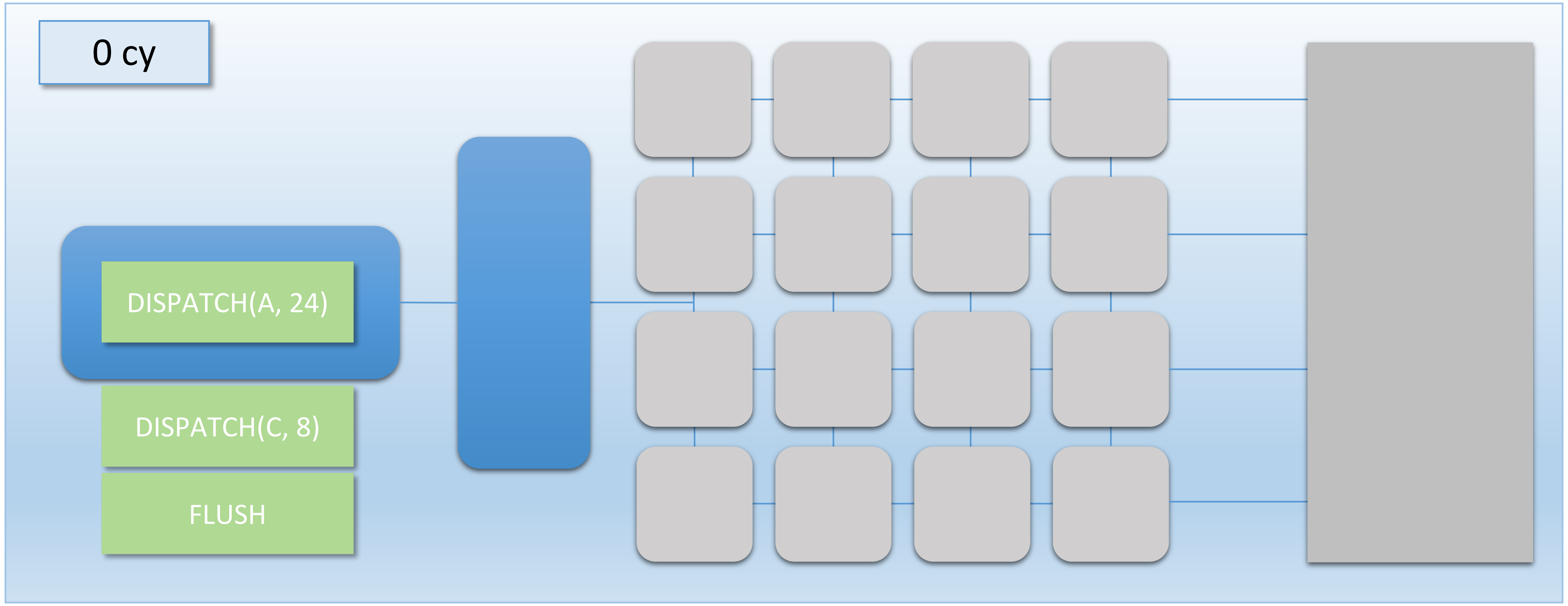
Overlapping Dispatches Example

- 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!

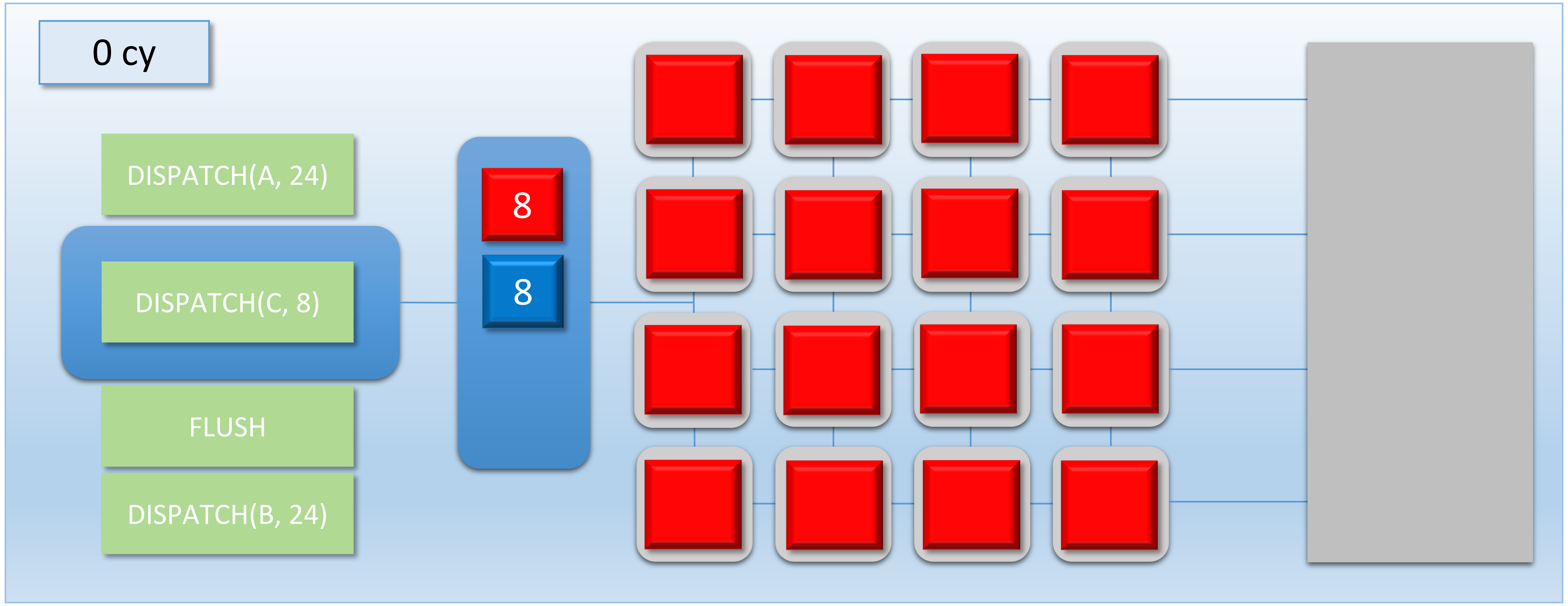
Insights From Overlapping

- 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)

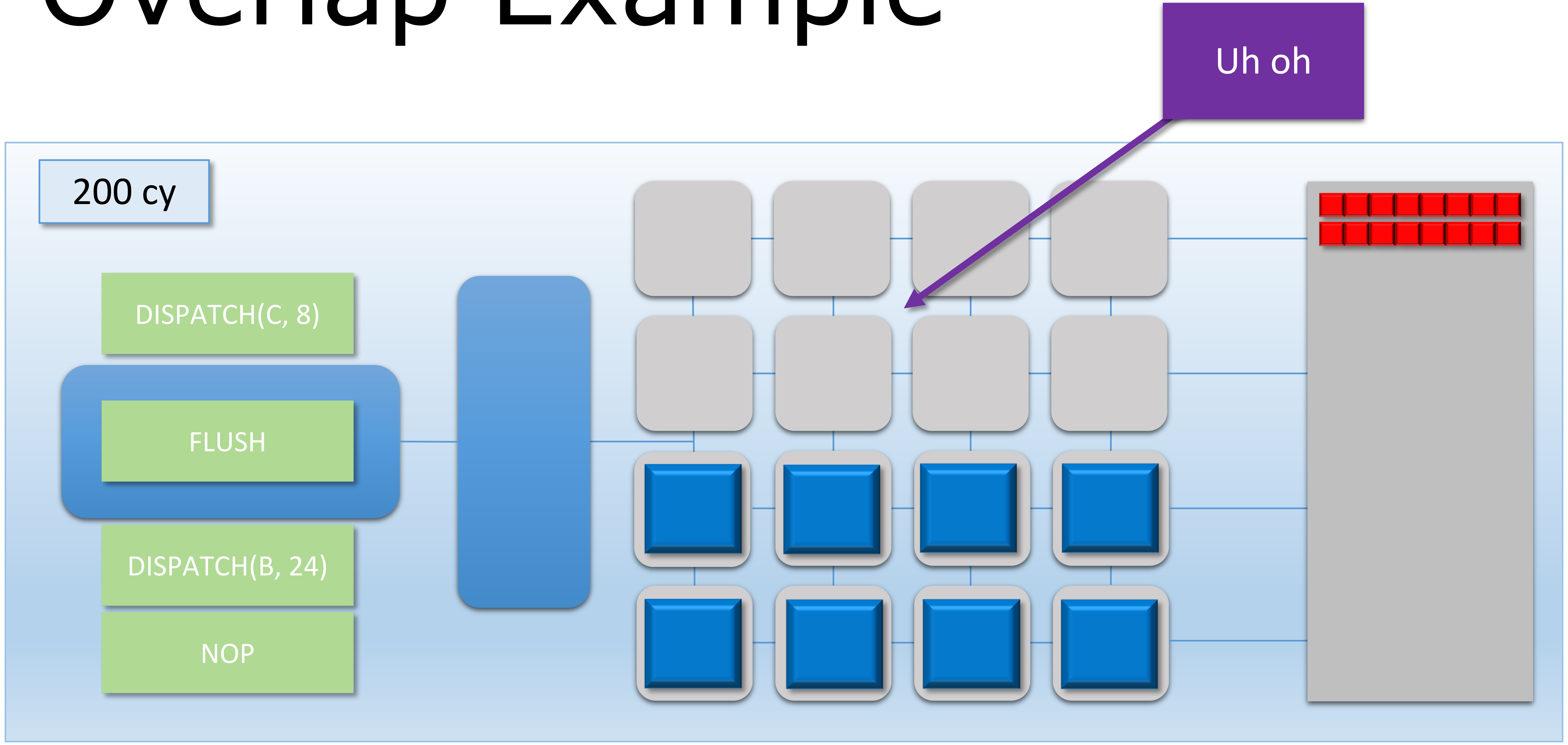
Bad Overlap Example



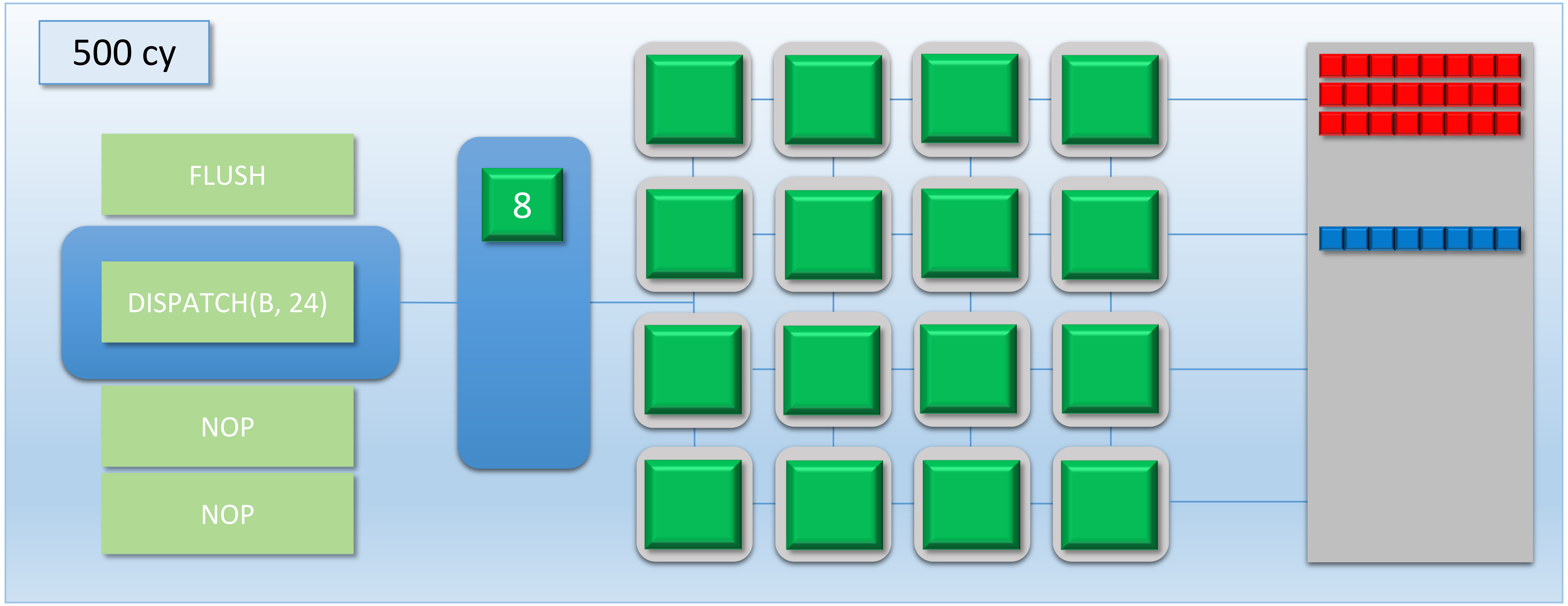
Bad Overlap Example



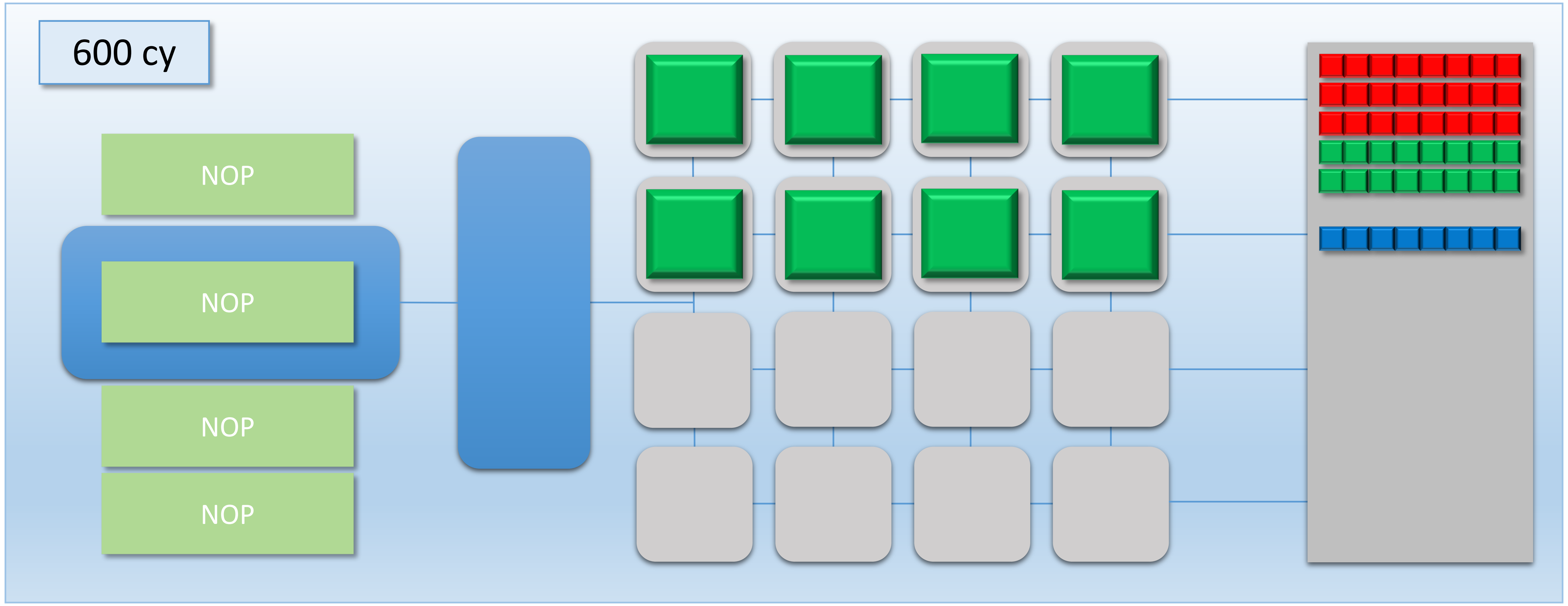
Bad Overlap Example



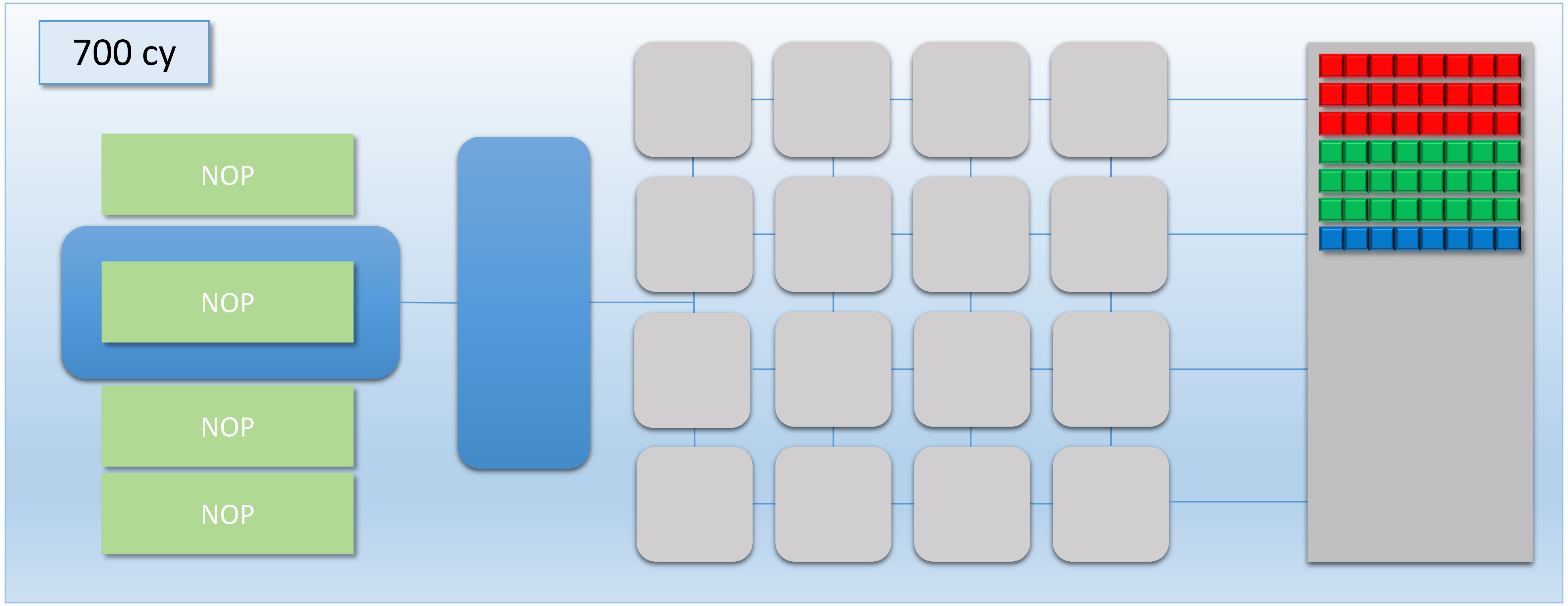
Bad Overlap Example



Bad Overlap Example



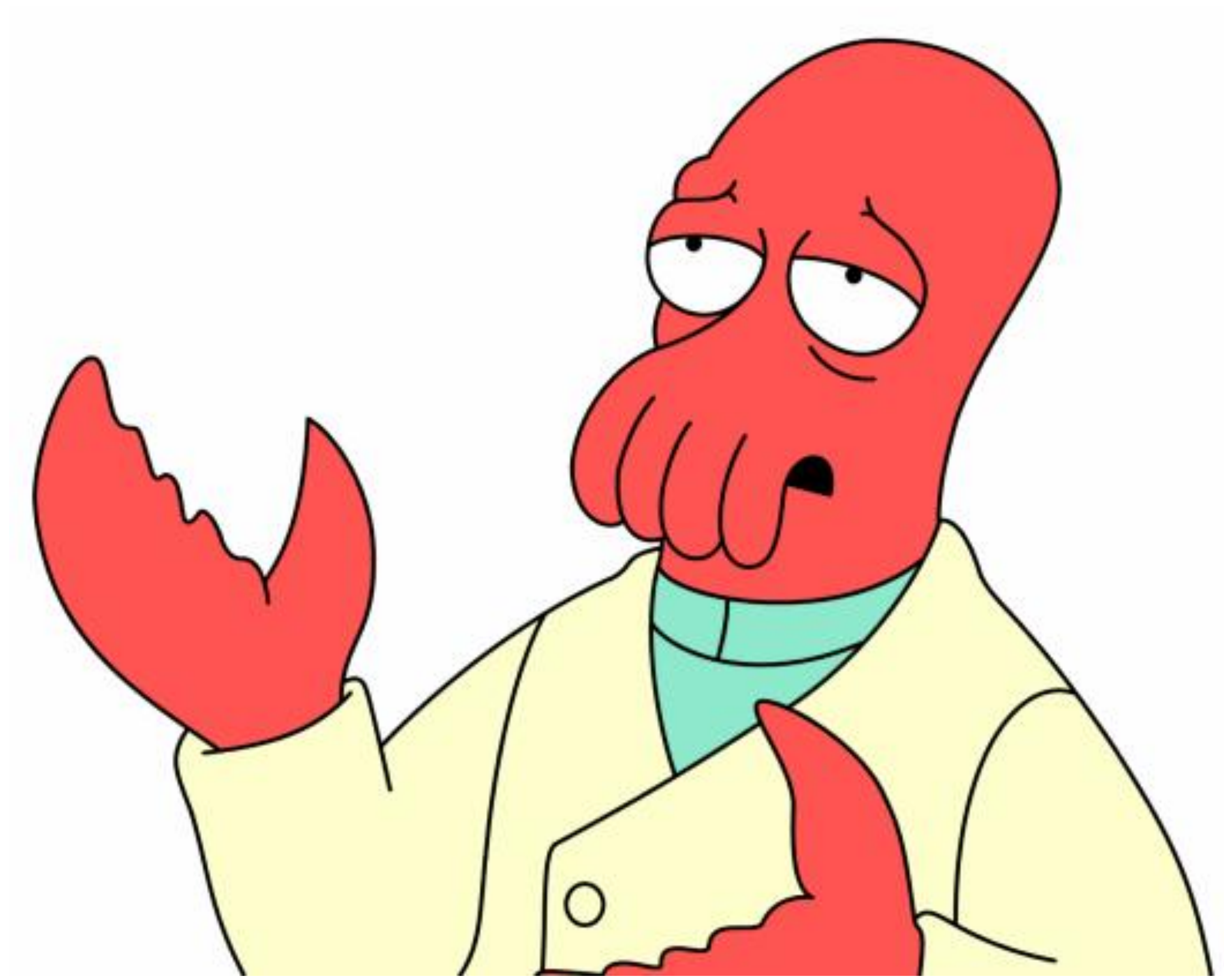
Bad Overlap Example



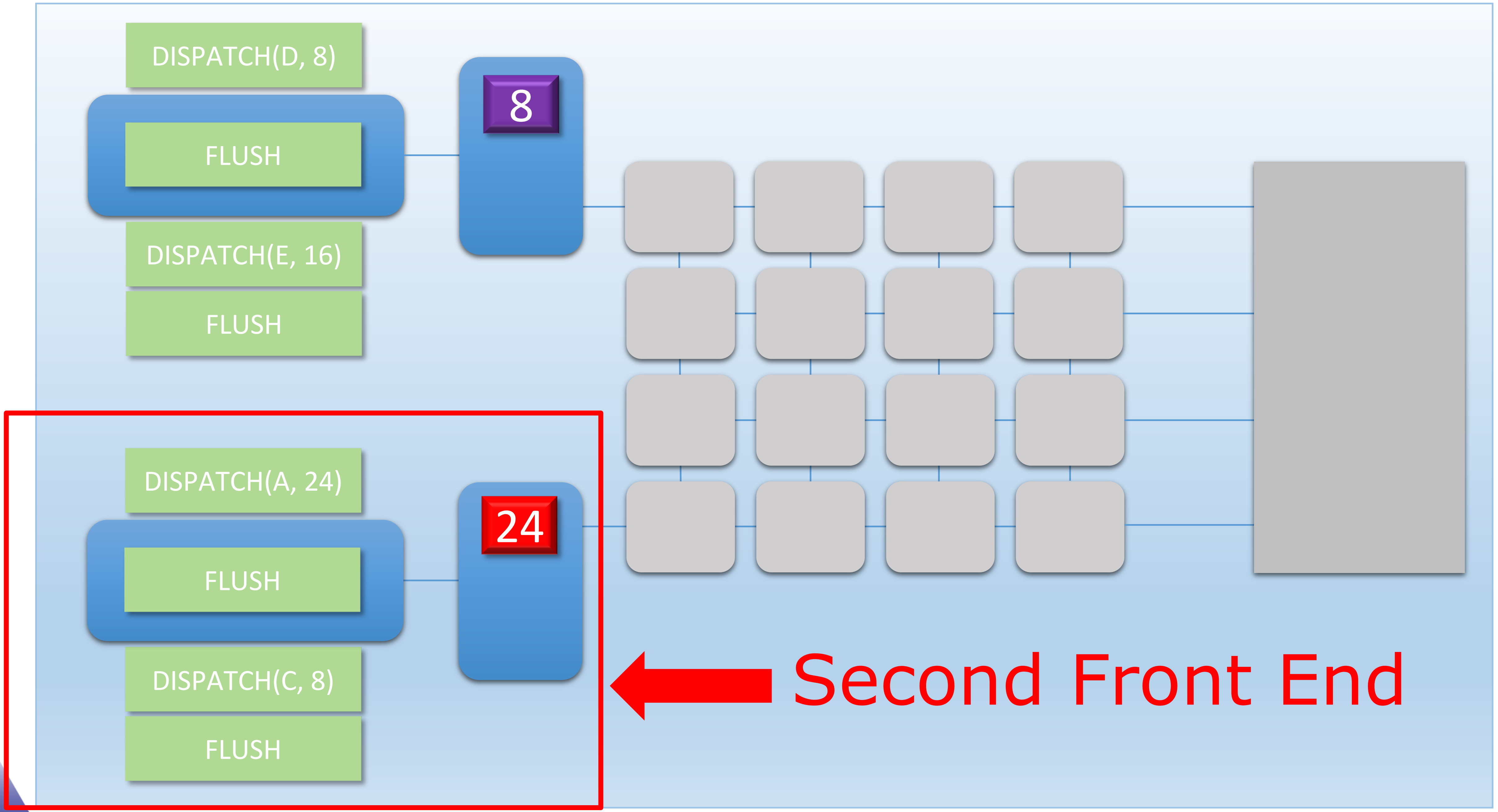
What Happened?

- 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



Introducing The MJP-4000

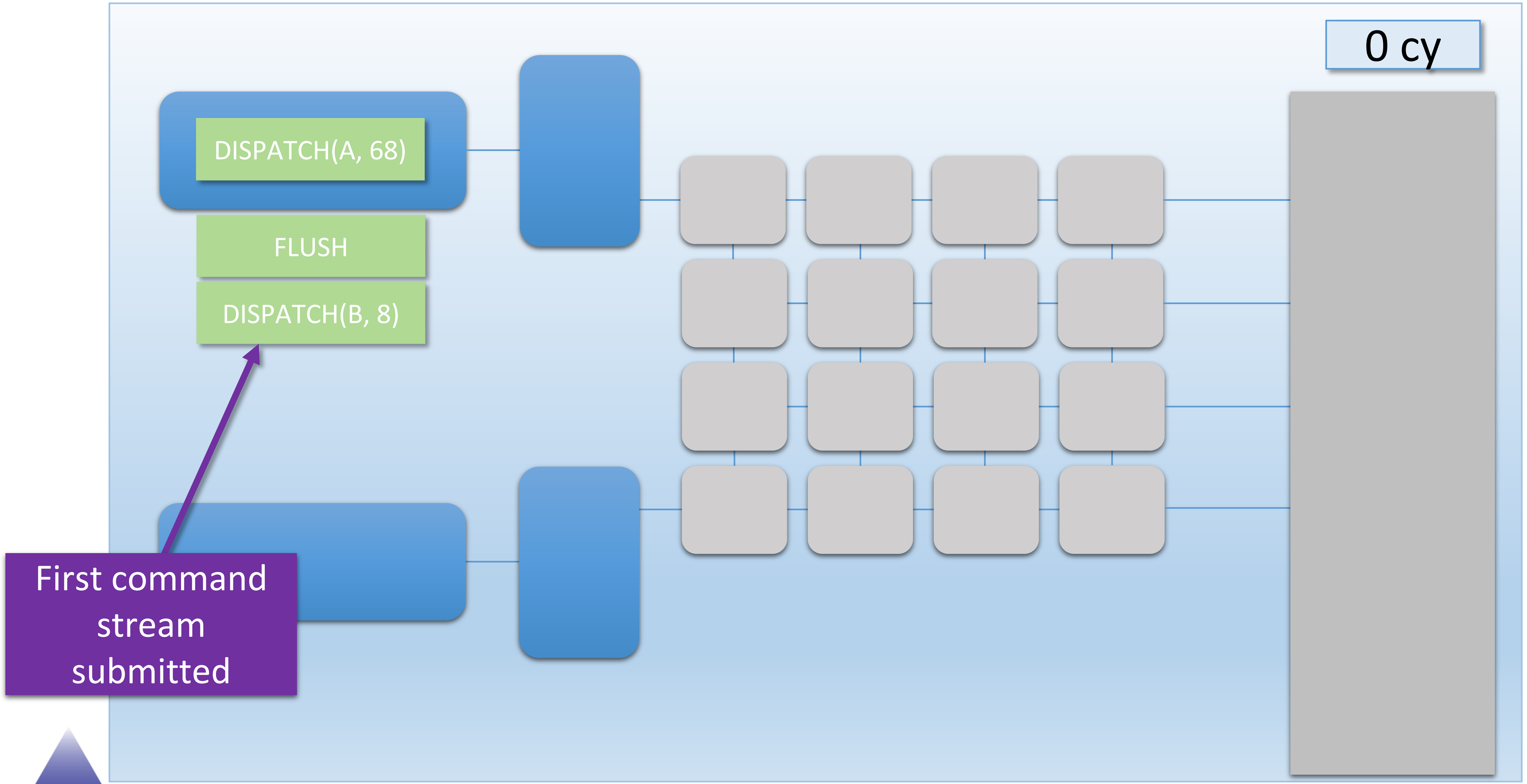
- Two front-ends
  - Two command processors for syncing
  - Two thread queues
  - Two $\boxed{\textbf{independent}}$ command streams
- Still 16 shader cores
  - Max throughput same as MJP-3000
  - First-come first-serve for thread queues

Dual Command Stream Example

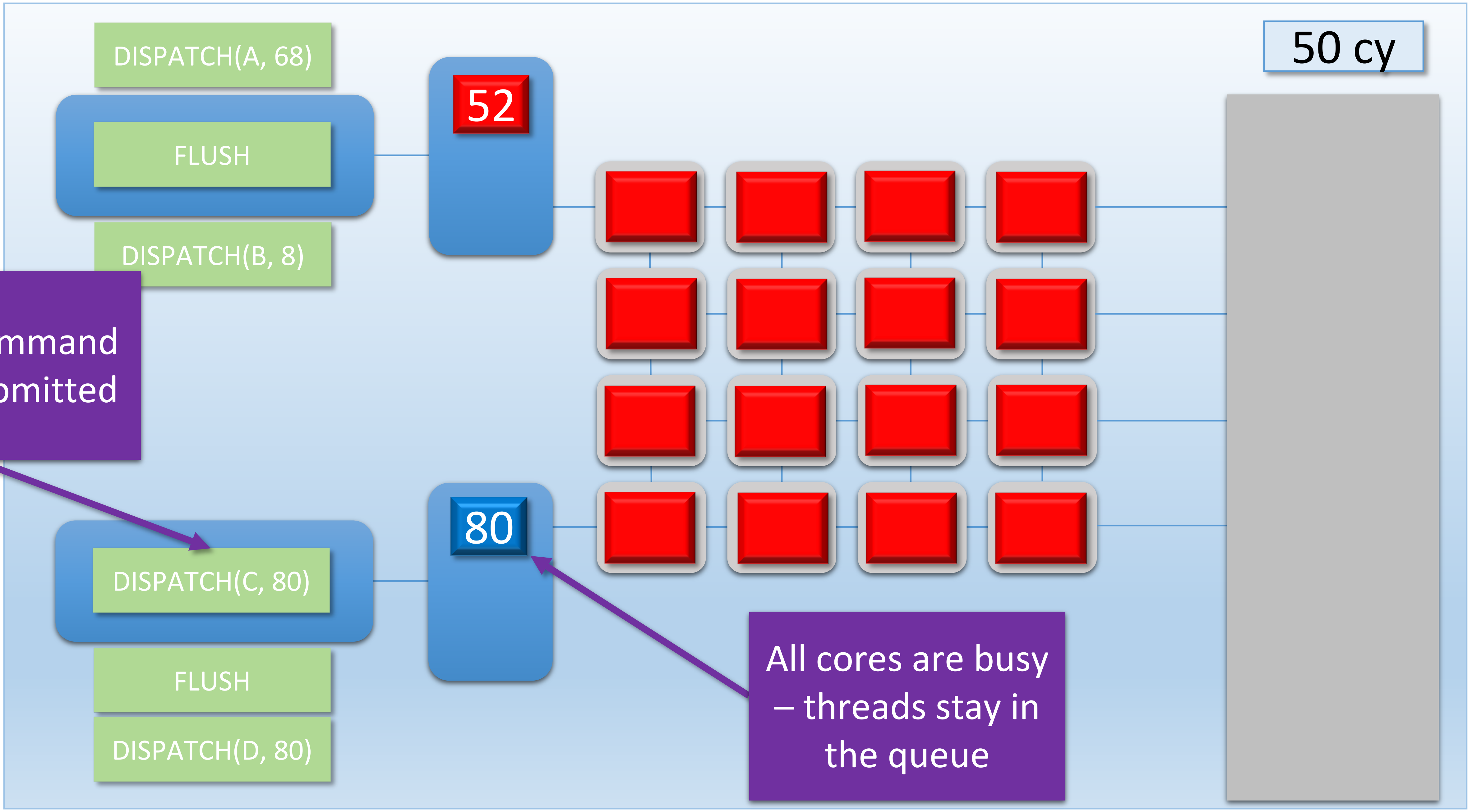
- Dispatch A -> 68 threads, 100 cycles
- Dispatch B -> 8 threads, 400 cycles
  - B depends on A
- Dispatch C -> 80 threads, 100 cycles
- Dispatch D -> 80 threads, 100 cycles
  - D depends on C

Independent command streams

Dual Command Stream Example



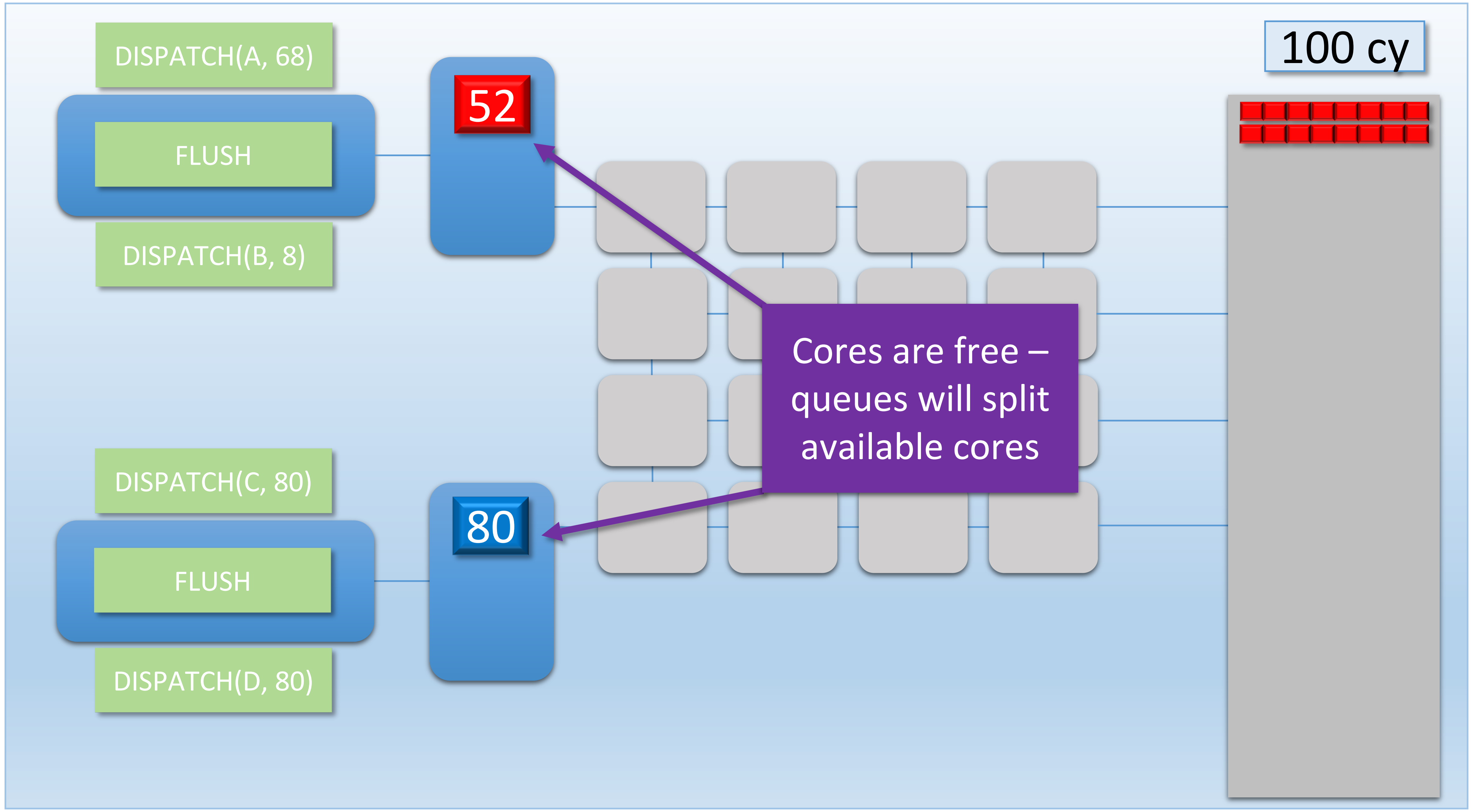
Dual Command Stream Example



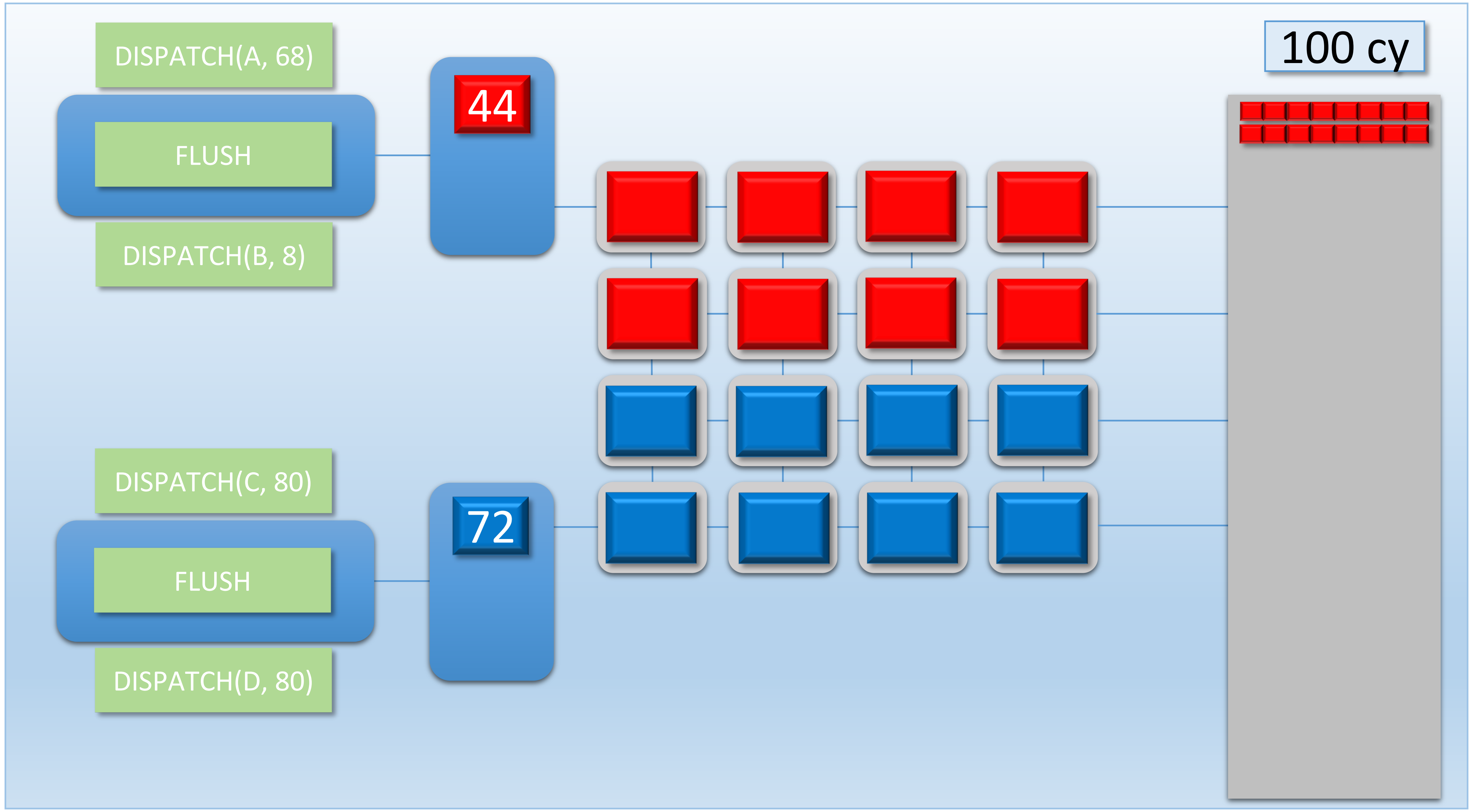
Second command stream submitted

All cores are busy – threads stay in the queue

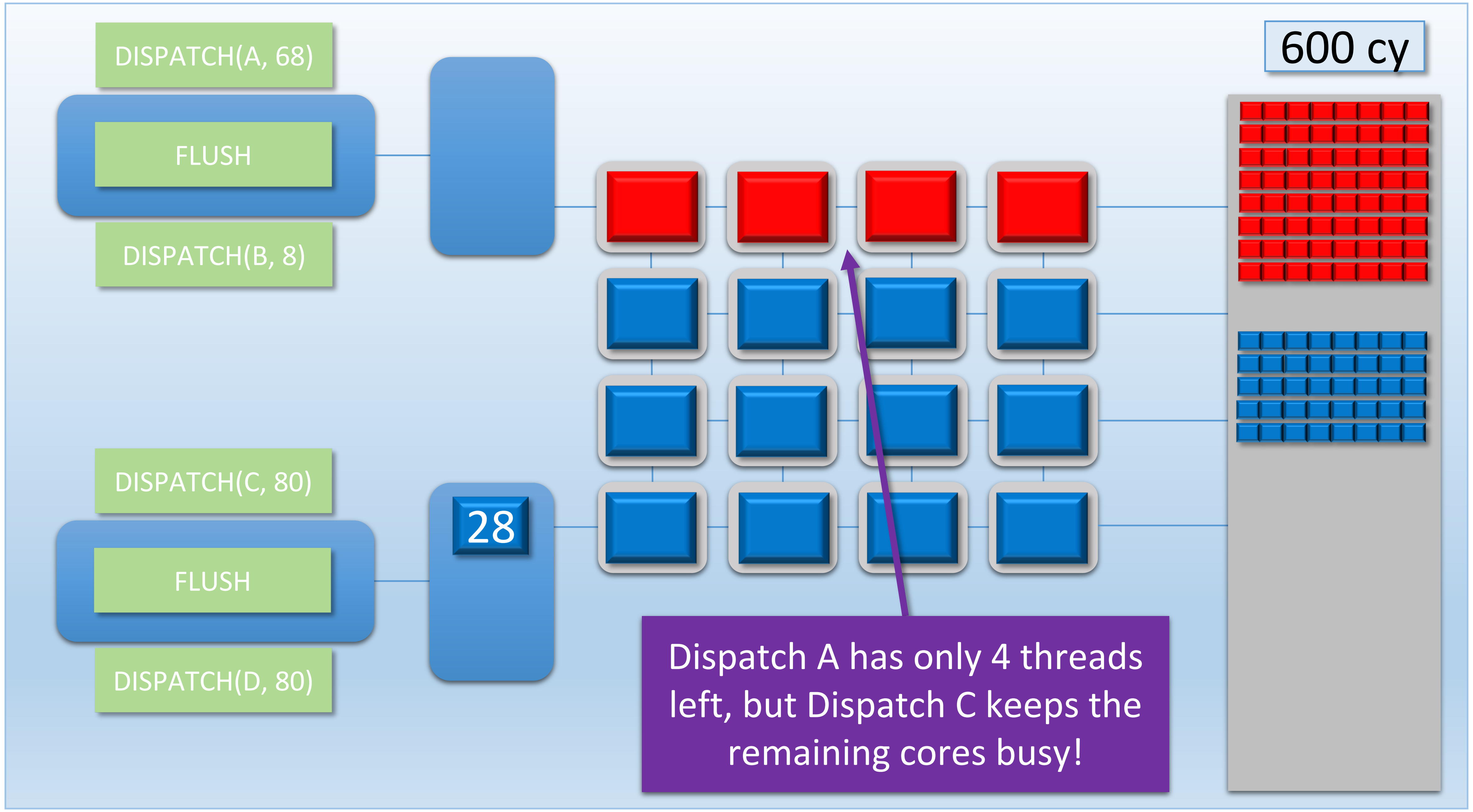
Dual Command Stream Example



Dual Command Stream Example

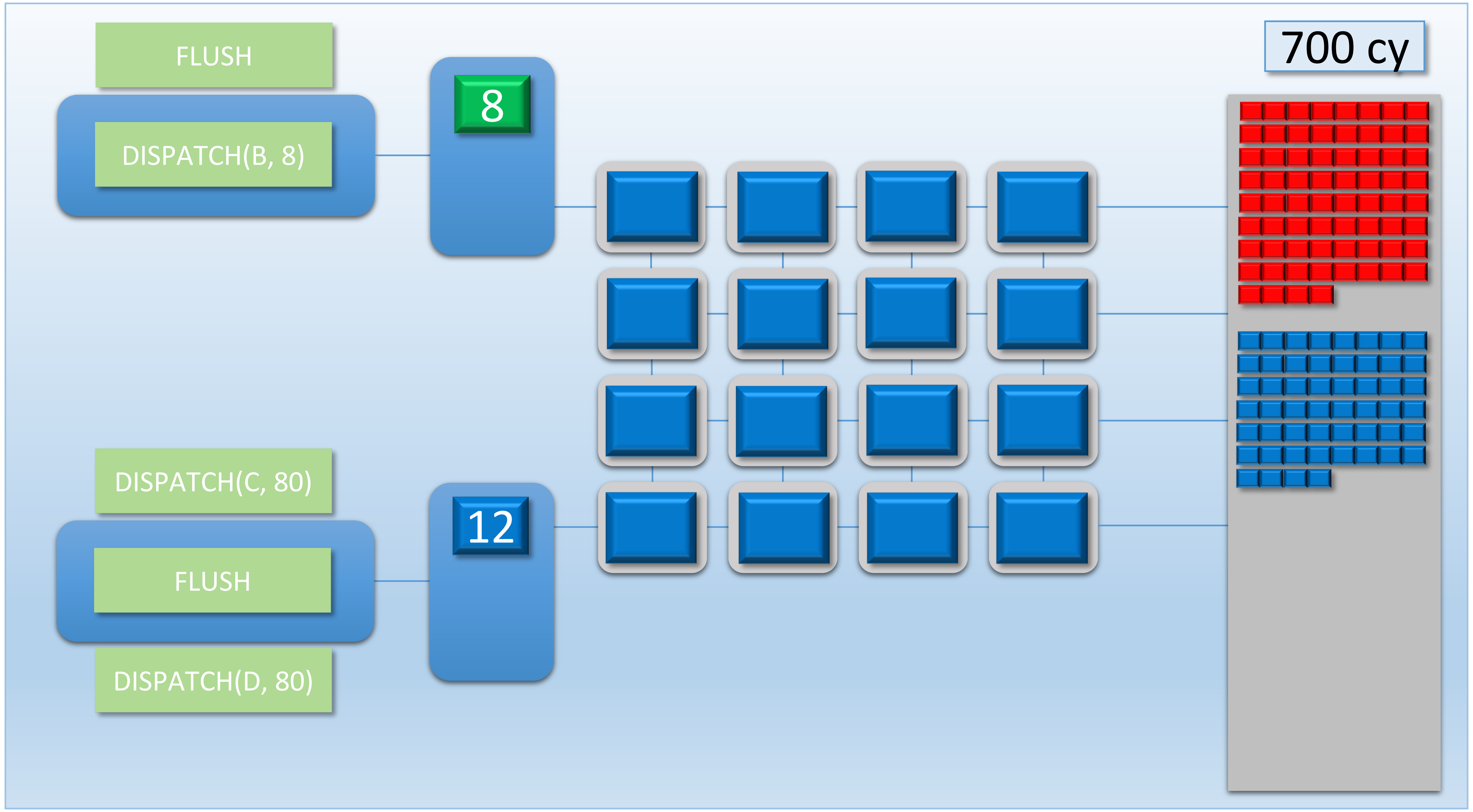


Dual Command Stream Example

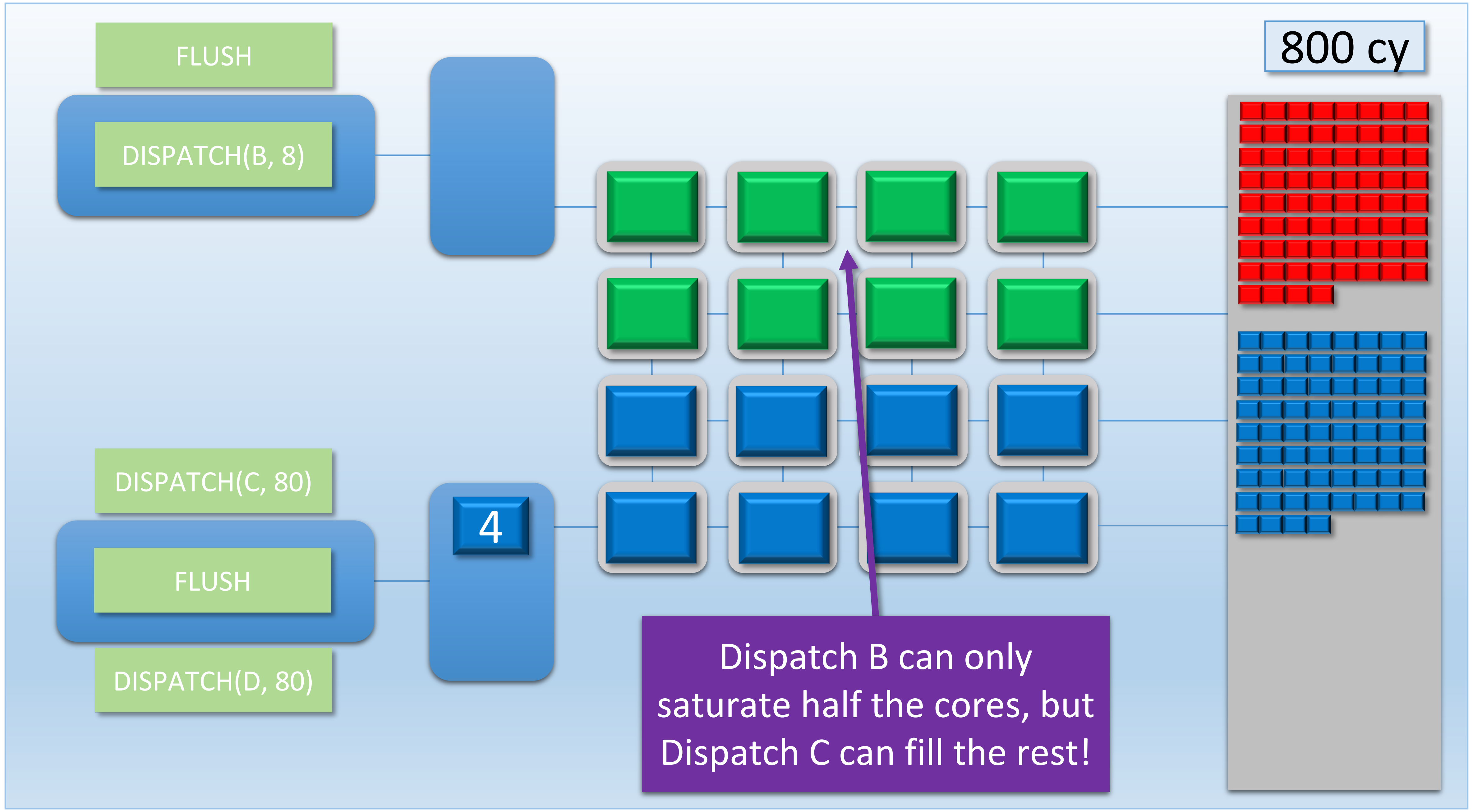


Dispatch A has only 4 threads left, but Dispatch C keeps the remaining cores busy!

Dual Command Stream Example

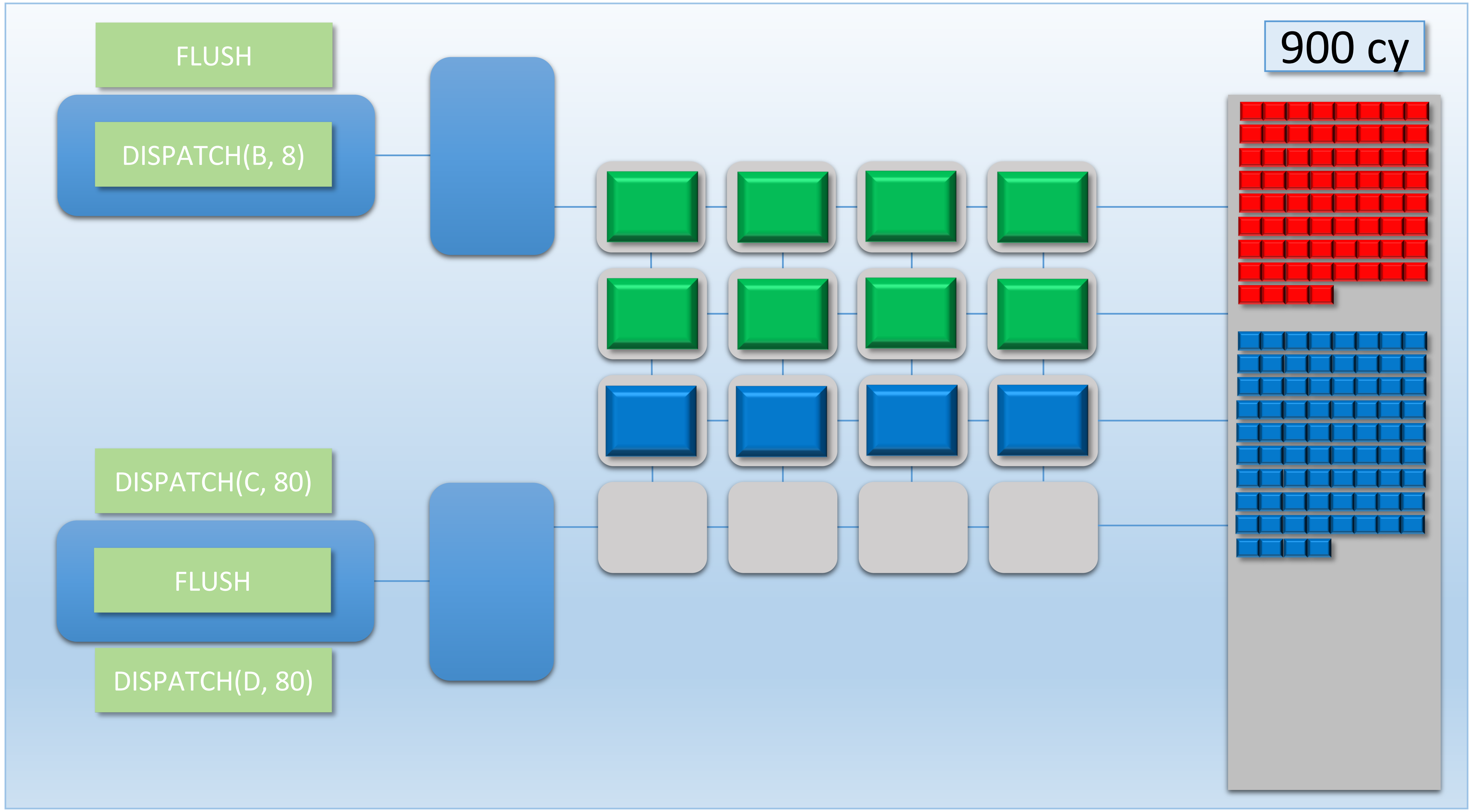


Dual Command Stream Example

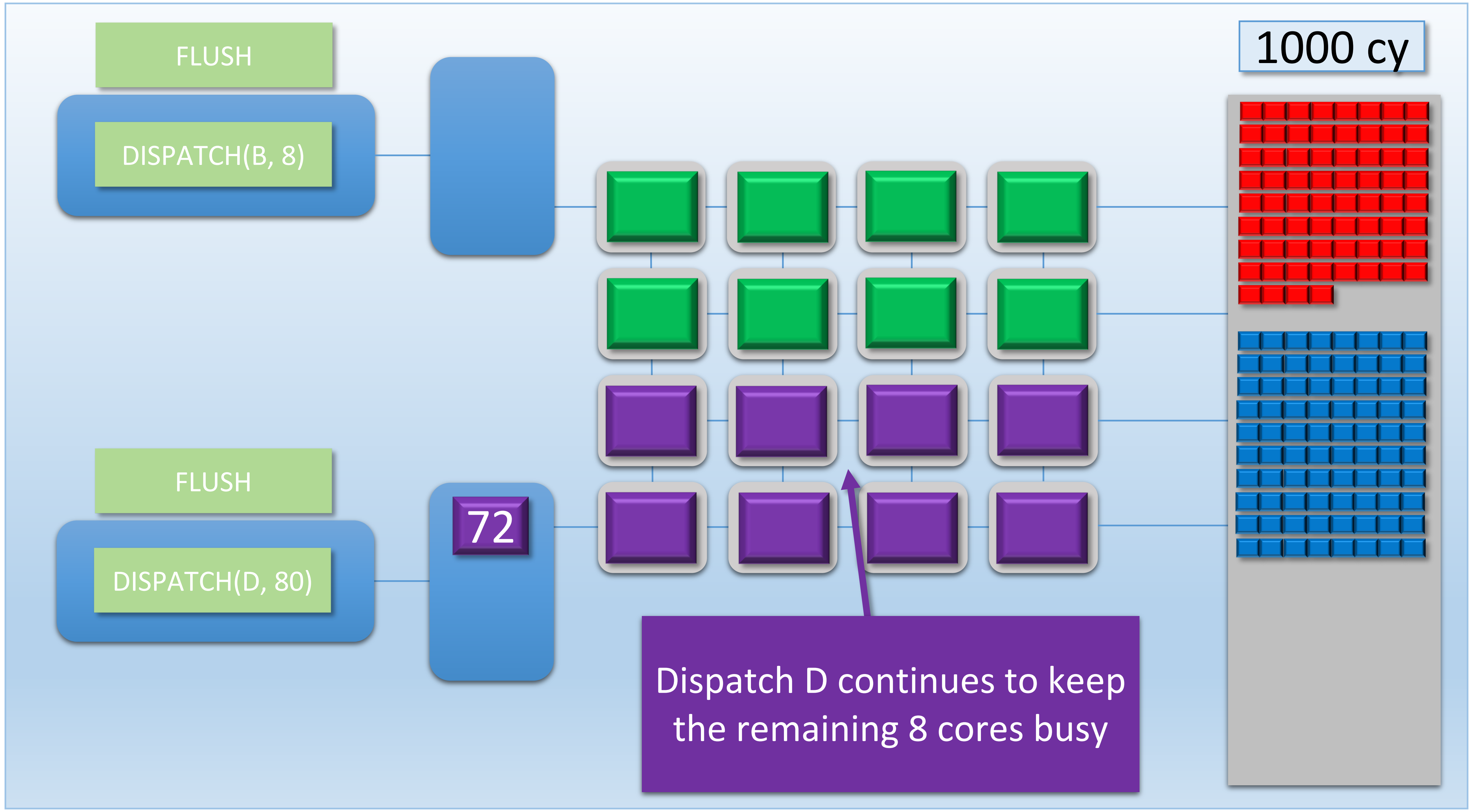


Dispatch B can only saturate half the cores, but Dispatch C can fill the rest!

Dual Command Stream Example

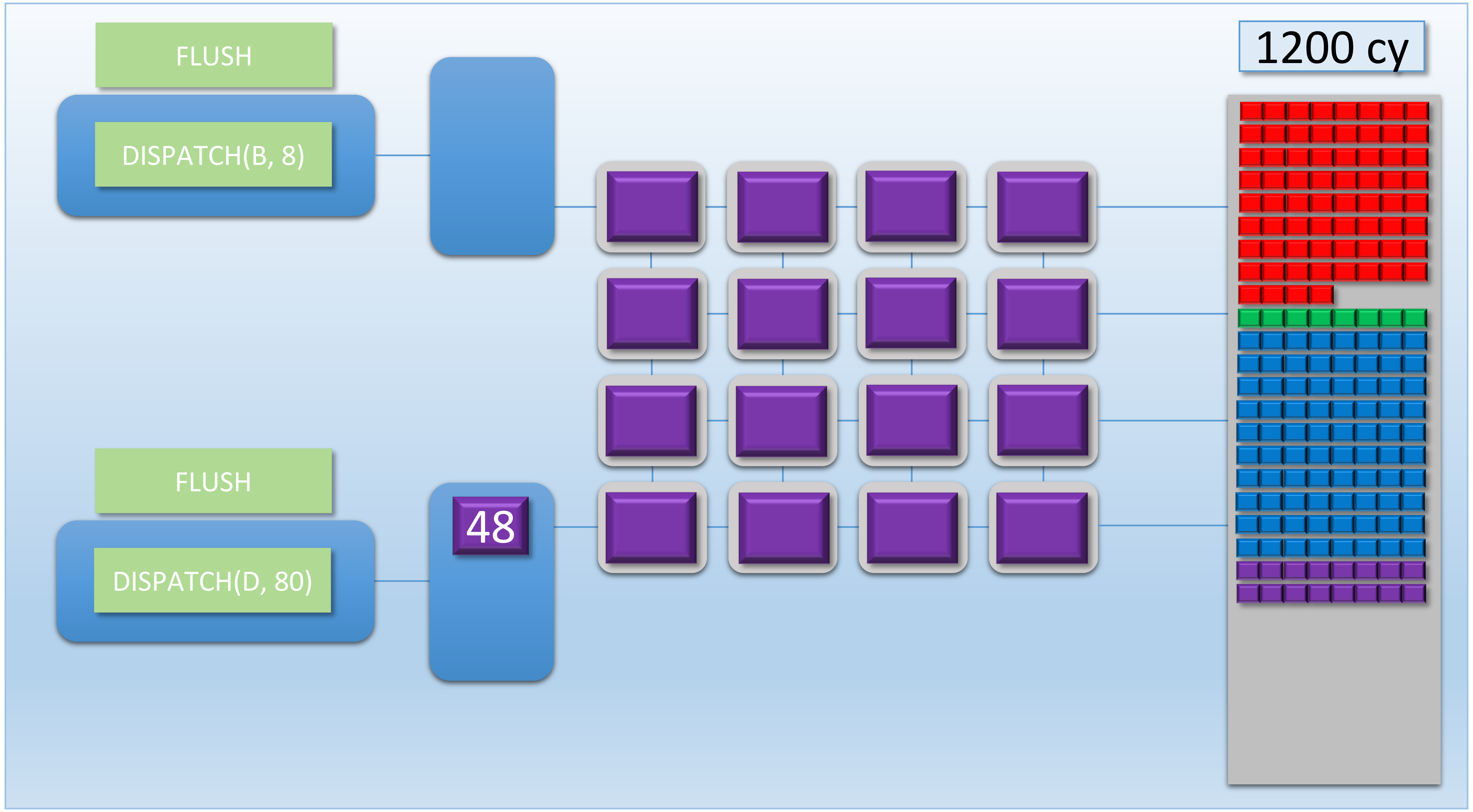


Dual Command Stream Example

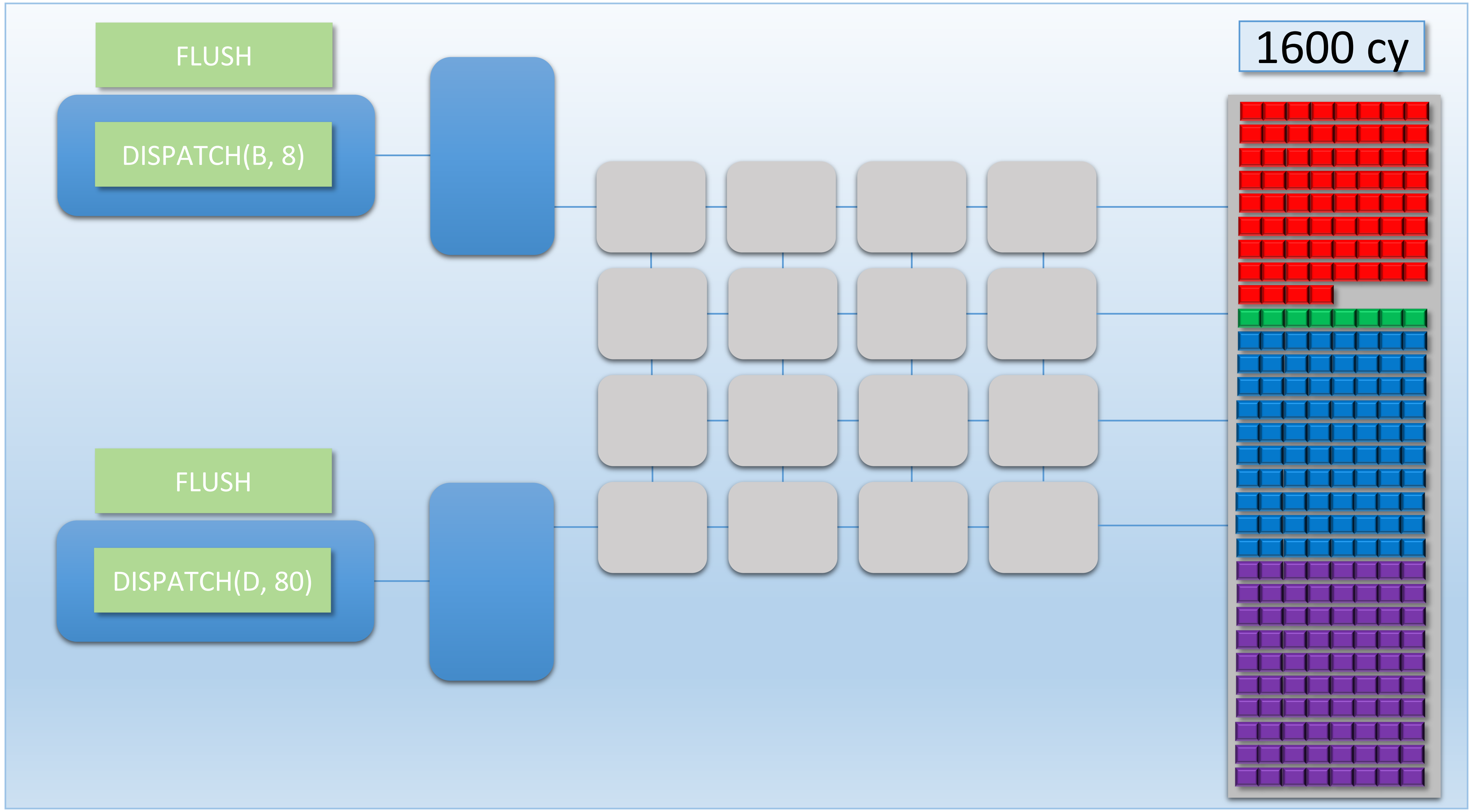


Dispatch D continues to keep the remaining 8 cores busy

Dual Command Stream Example



Dual Command Stream Example



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

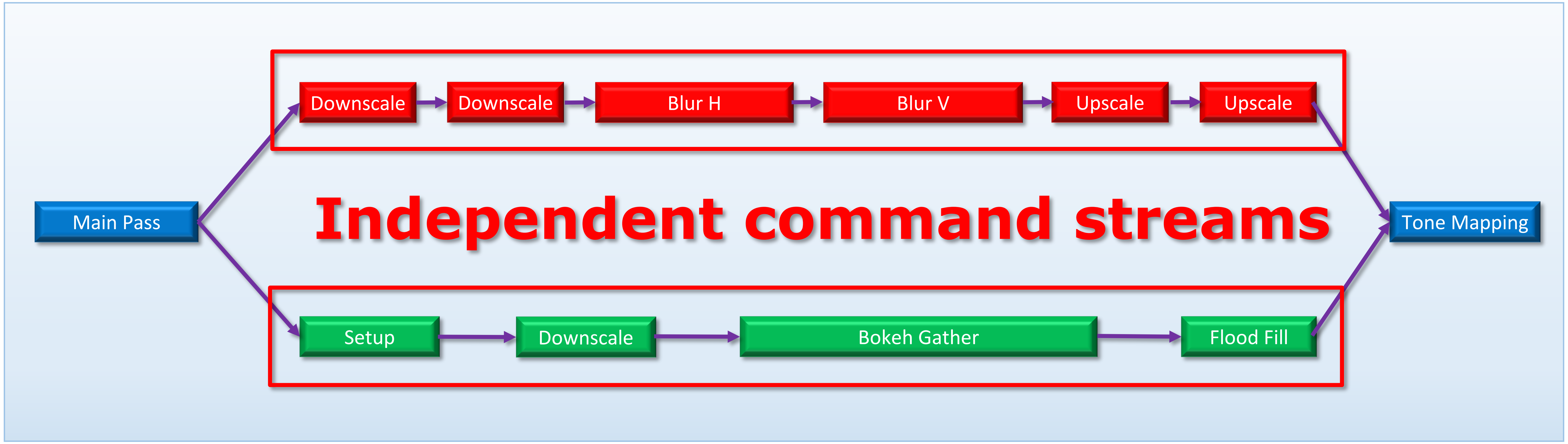
Even Better For Real GPUs!

- 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

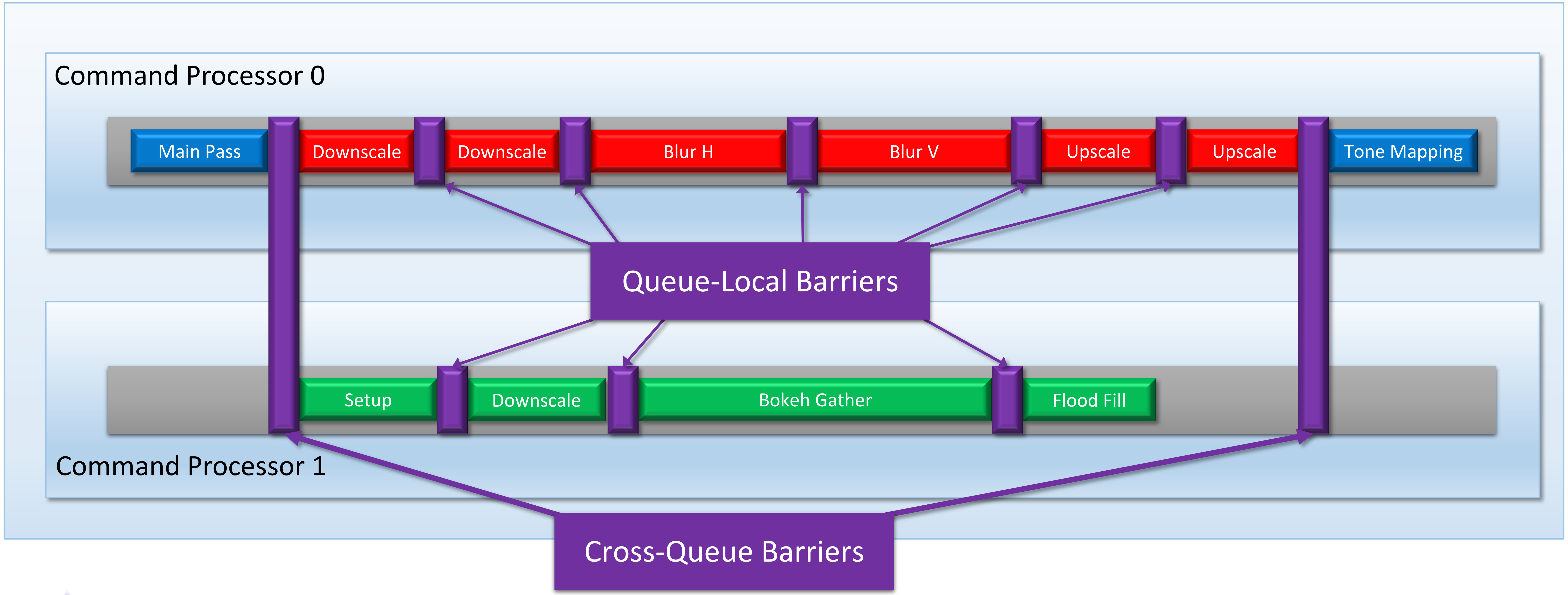
Thinking in CPU Terms

- 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



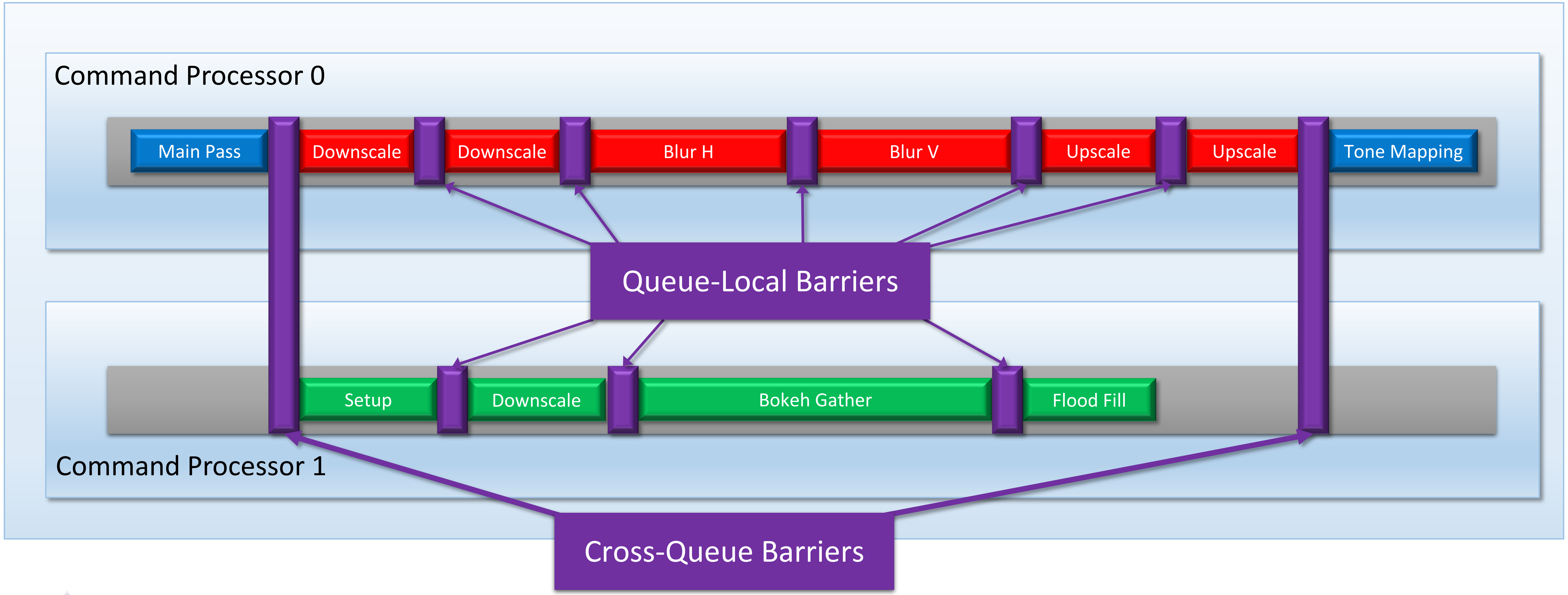
Real-World Example: Bloom + DOF



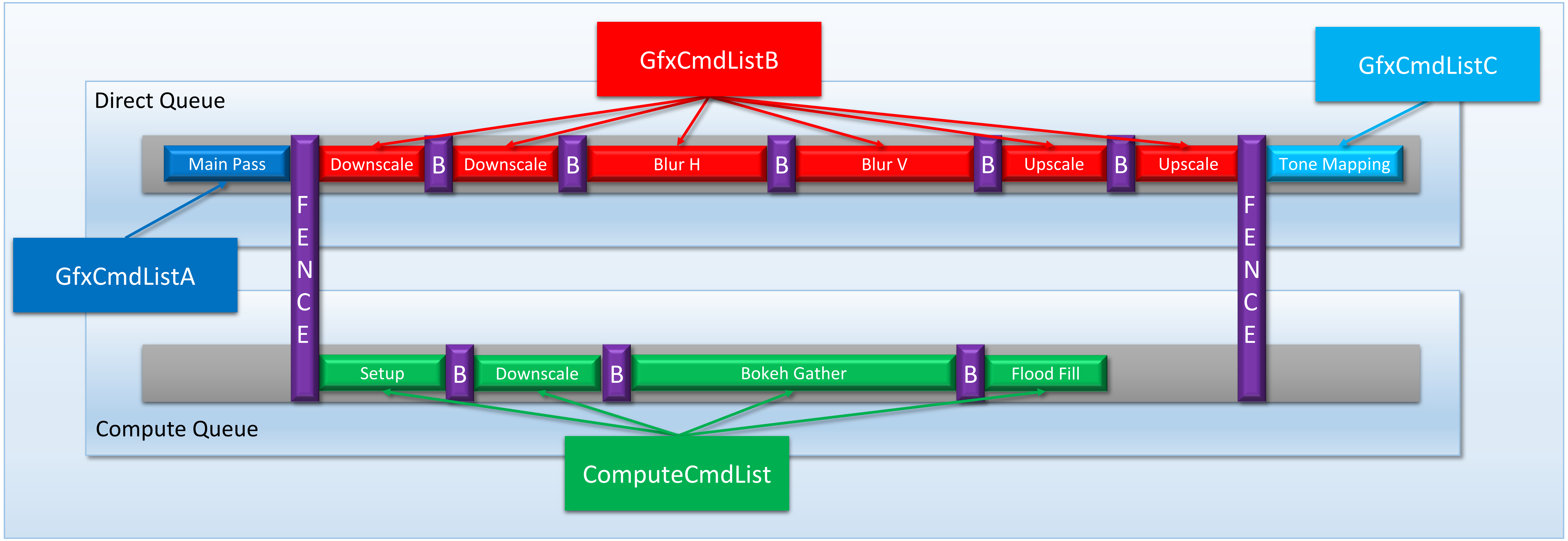
Submitting Commands in D3D12

- 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 ≈ 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 ***<u>may</u>*** 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