Hey there, thanks for downloading this talk! I've added notes to all of the slides that roughly match what I said during the actual presentation at GDC. So please read along with the notes if you want to get the full experience. ☺

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!

So real quick, I just wanted to introduce myself and tell you a bit about what I do and why I'm here today. I've been an engine programmer at Ready At Dawn Studios for the past 9 years or so, which is an independent developer located in Irvine, California. I mostly work on graphics/rendering things, but also do quite a bit of general work on our in-house engine. I've been the lead of our engine team for the past 5 out of those 9 years.

In case it wasn’t already obvious, I really like learning about and discussing GPUs, as well as the APIs that we use to talk to them. Over the years I’ve done a lot of blogging about GPUs and graphical techiniques on my personal blog called “The Danger Zone” (https://mynameismjp.wordpress.com), and I’m also pretty active on Twitter. I have a public repos on GitHub as well, which are mostly samples demonstrating graphical techniques using DX11 and DX12. If you do know me from the internet, then you probably 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?

So why did I get you all in this room today? As you can probably guess from the title of this talk, I'm going to try to give you all an introduction to GPU synchronization using the most gentle and straightforward explanations that I can. Really this is intended for people that aren't already experts on this subject, and is intended to understandable even if you're relatively new to graphics programming. So I'm really going to try to start from the basics by

explaining what the heck a barrier is, and then moving on to explaining why we even need them in the first place. Afterwards I'm going to also try to explain how they generally work on currently-available GPUs, in particular when it comes to thread synchronization. Finally I also want to help give you all a rough intuition on how (and why) barriers have an adverse effect on performance, which should help you structure things more optimally when using APIs that require manual barrier usage.

Barriers in D3D12/Vulkan

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

So, barriers are a thing that we have to deal with now. As far as graphics APIs are concerned they're a (mostly) new concept that came along with the newer "explicit" APIs like D3D12 and Vulkan. If you've done any programming with either of those two APIs, you've probably realized that barriers can really annoying to deal with. I mean we didn't have to deal with this at all in D3D11 or OpenGL, so it's a bit head-scratching that we suddenly have to

do all of this manual book-keeping just to get correct results on the screen. They're also notoriously hard to get right: this is maybe the 4th year that people have gotten up on this stage to talk about barriers and how to get them "right", and the validation layers seem to always be yelling at us for messing them up. From previous presentations you've probably already heard that using barriers will have an affect on your GPU performance, meaning that even if you are getting correct results you still want to optimize them in order to get the best possible frame times. But it may not be obvious at all as to *why* these barriers hurt your performance, and what the exact impact will be. We're going to go through that bit later on in this talk.

CPU Thread Barriers

- Thread sync point
- "Wait until all threads get here"
  - Spin wait
  - OS primitives
- Barrier is a toll plaza

Before we talk about the GPU side of things, I thought it might be helpful to talk about barriers as they apply to CPUs. That way those of you with experience in multicore CPU programming can see how the concepts carry over.

One kind of barrier you use on the CPU is called a "Thread Barrier". These are essentially a thread synchronization point, which you use to ensure that all threads have

reached a certain point in your code. They’re generally implemented by having threads wait around until all other threads reach the same point, typically by spinning on shared variable or by using an OS-level primitive such as a semaphore. Some very simple code using a thread barrier might look something like this:

```
void ThreadFunction()
{
    DoStuff();

    // Wait for all threads to hit the barrier
    barrier.Wait();

    // We now know that all threads called DoStuff()
}
```

The best metaphor I can come up with for this is a toll plaza, except one that’s very slow and causes all of the cars to stop instead of just passing right through.

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

The other kind of barrier that you see a lot in CPU code is called a “Memory Barrier”. These are all about ensuring that low-level CPU memory operations (reads and writes) happen in the order that’s required for your code to work correctly across multiple cores. Probably the most common example is the “mailbox problem”, which is where one thread stuffs some data into one variable (the mailbox) and then sets another variable afterwards indicating to

another thread that the mailbox is full:

```
uint32_t DataIsReady;
SomeStruct Data;

void ThreadFunc()
{
    WriteToData(&Data);

    // Make sure that all previous
writes to Data are complete
    // before the write to
DataIsReady is visible

std::atomic_thread_fence(std::memory_
order_release);

    DataIsReady = 1;
}
```

For various hardware-specific reasons, processors with so-called “weak memory models” (such as ARM or PowerPC) won’t guarantee that the write to the “ready” flag will be seen *after* the write to the mailbox when viewed by code on another core. To make sure this happens, you need to put in a memory barrier. These barriers also inform the compiler that the order of writes is important should be preserved, since the compiler might re-order things if the results are being read by another thread. In this case the visual metaphor is a doggie gate, since a barrier effectively keeps memory operations on one side of the barrier so that they don’t cross over.

Some links to look at if you want to learn about memory barriers:

https://docs.microsoft.com/en-us/windows/desktop/DxTechArts/lockless-programming
https://channel9.msdn.com/Shows/Goin

g+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2
https://preshing.com/20120612/an-introduction-to-lock-free-programming/
https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html (chapter 15)

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!

So what's the common thread between these two kinds of barriers? The answer is that they're both needed to correctly handle data dependencies. A dependency naturally occurs when one piece of code writes some data, and another piece of code reads from that data. In that case we would say that the second piece of code is dependent on the first bit. Whenever this kind of dependency occurs we need to make sure that the results are *visible* to

the dependent code, which essentially means that the dependent code reads exactly the same data that was written.

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

When you're writing single-threaded code you're naturally producing dependencies, but generally you don't really have to think about them very much. For instance you might compute an offset into an array, then use that offset to look up some data from that array. You don't really need to think about this dependency most of the time since this single-threaded case can be handled by the compiler and CPU without manual intervention. The compiler is actually

capable of identifying these sorts of dependencies automatically when its looking at a single thread, which means it can do whatever is necessary to make sure that you get the correct results. In terms of low-level memory operations even architectures with weak memory models will produce the correct ordering as long as everything is running on a single core, so you don’t need to worry about memory barriers. Basically this is what I would call “easy mode”, since everything just works without you needing to do anything special.

Multi-Threaded Dependencies

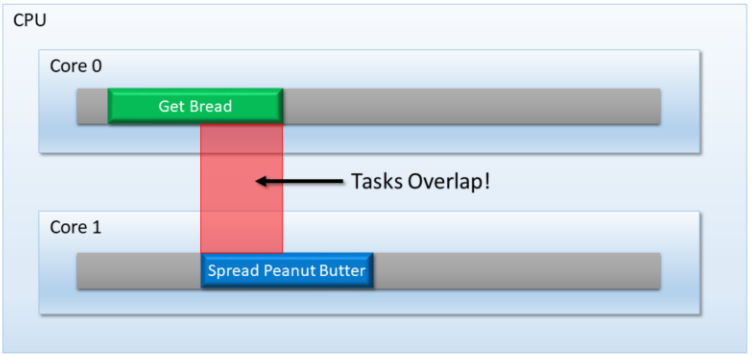
- Dependencies no longer visible!
  - Arbitrary numbers of threads
  - Free-for all memory access
- CPU mechanisms break down
  - Per-core store buffers and caches
- Everyone has failed you
  - You're on your own

Once we get to the multi-core world, suddenly everything is much harder. The main problem here is that once you start writing multiple threads, the dependencies between those threads are no longer visible to the compiler. In general threads are allowed to read and write whatever memory they want within a process's virtual address space, so it's not really practical to expect the compiler to magically be able to sort all of that out for you. Multi-core execution is

also when CPU mechanisms for ensuring ordering start to break down, and so now you have the issues that come about on architectures with weak memory models. The specifics are a bit too complex to get into here, but some of the reasons for these ordering issues include HW-specific optimizations include per-core store buffers and other specialized caches.

Basically what I'm trying to say here is that once you're writing multiple threads everything else has completely failed you, and now it's totally on you to start putting barriers and things in the right places if you want to get the results that you expect. In other words you're Link starting out into a world of scary monsters, and all you've got to defend yourself is the atomic library from the standard library.

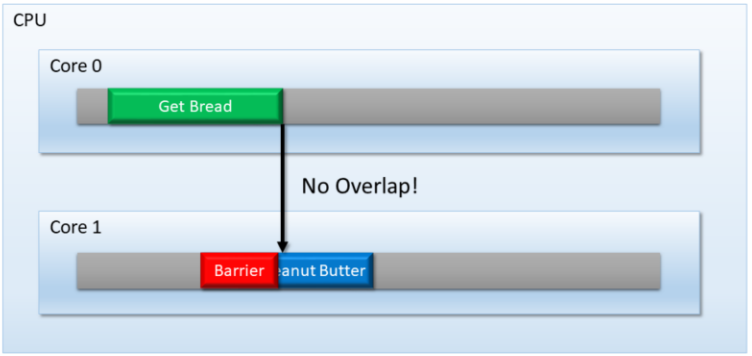
Task Dependencies



So let's look at a really simple example of a multi-core task dependency. On the first core we have a "Get Bread" task, where we need to grab some bread out of the cabinet and put it on a plate. Then on the second core where we have a "Spread Peanut Butter" task where we put peanut butter onto that break we just retrieved. The peanut butter task is naturally dependent on the output of the bread task, which is bad since the peanut butter task

starts up before the bread task actually finishes! We would say that these two tasks overlap here, since they occupy the same time period when viewed on a timeline like we have here.

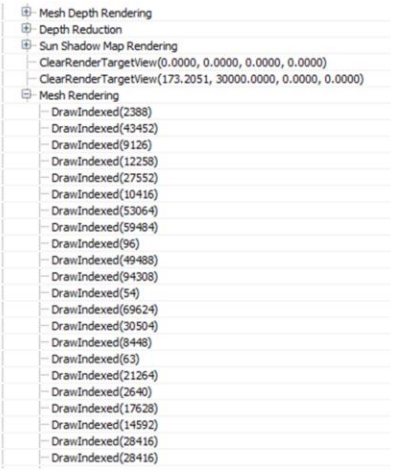
Task Dependencies



To make sure that we get the correct results in this case, we have to stick in a thread barrier that will wait for the bread task to completely finish before allowing the peanut butter task to start. So if we do that, we see the peanut butter task slide over until it no longer overlaps with the bread task. Now we should actually spread the peanut butter on the bread, instead of spreading it onto an empty plate. ☺

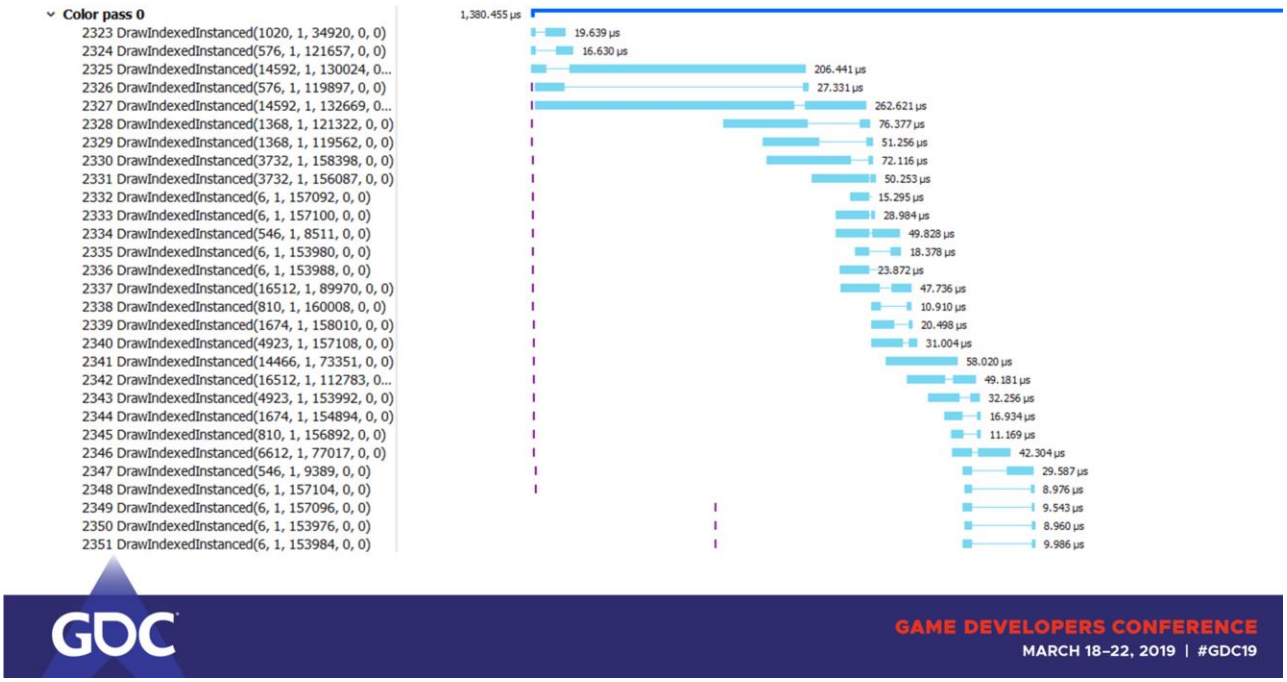
GPU Parallelism

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



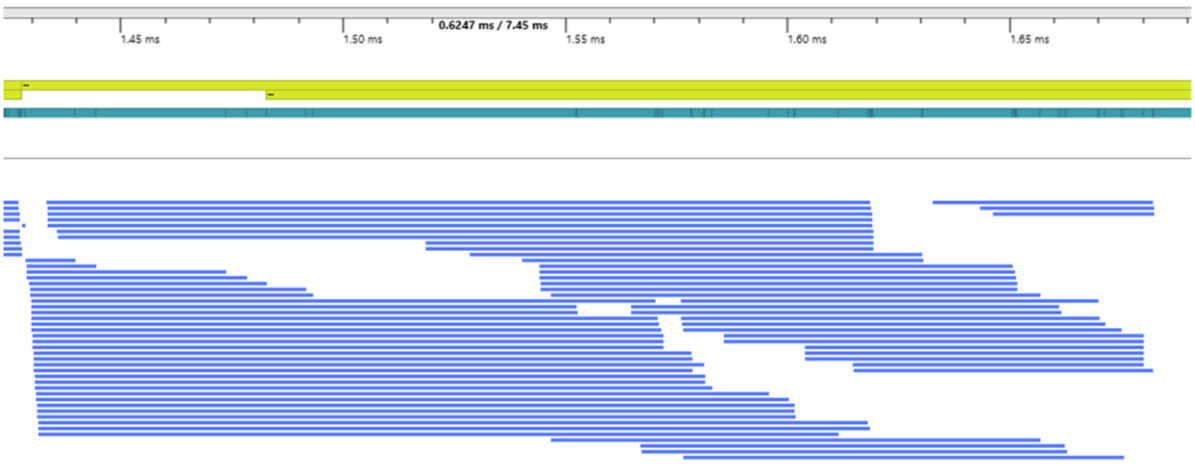
So lets finally switch over to GPU land and talk about what goes on there. At some point or another you've probably heard that GPUs are parallel machines with lots of cores on them. Despite that, you may still think of it as a serial machine in terms of how it processes the work you give it. I mean we typically give it commands that look like what we have on the right, where we expect each Clear or Draw or SetRenderTarget to happen

sequentially. In fact if you use a GPU debugger such as RenderDoc (pictured here), they will typically let you click on each command to show you the results of each command one after another. It’s important to know that this is *not* at all how a GPU typically works! There’s actually all kinds of parallelism going on under the hood despite the serial nature of rendering APIs, a lot of which is possible because the drivers and the HW itself has of “magic” in there to extract parallelism for you.



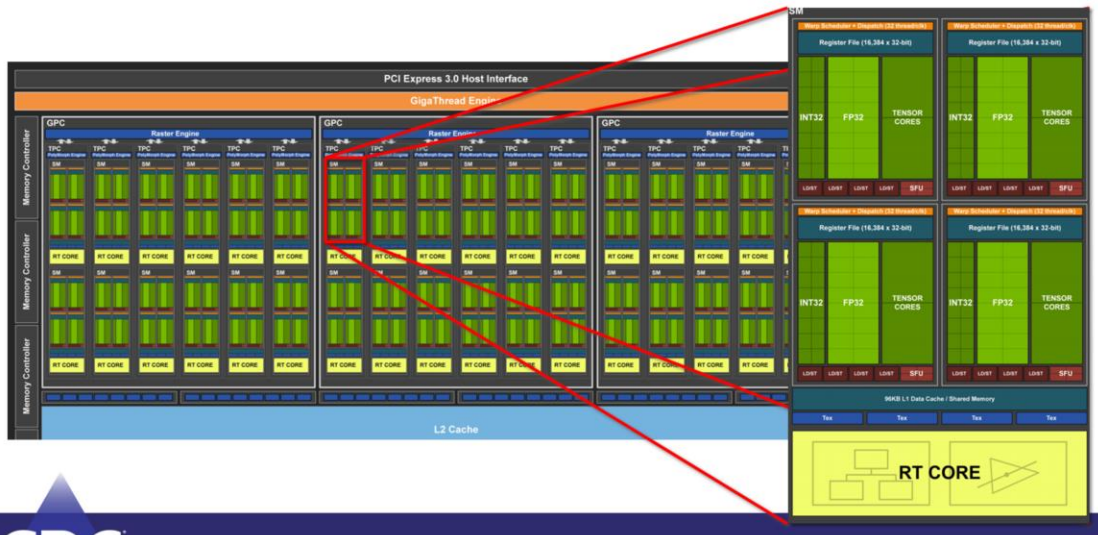
Here we have a screenshot from AMD’s Radeon Graphics profiler, which is an awesome tool for checking out the performance characteristics of your app when running on an AMD GPU. This is a “timeline” view, which is showing when each command actually starts executing on the GPU, as well as when it actually finishes. You can see here that these draw calls are definitely not executing sequentially, they’re actually overlapping all over

the place (especially the small draws).



This is a similar view from a capture taken with PIX for Windows, which is also showing when a bunch of draws begin and end. This one was taken on an even bigger GPU than the RGP capture from the previous slide, and so you can see there’s even more overlapping going on here.

GPUs are Thread Monsters!



The reason why we have all of this overlapping going on is because GPUs are big thread monsters! Take a look at this high-level diagram of Nvidia's latest Turing architecture. Each one of these blocks labeled "SM" is what they call a "Streaming Multiprocessor". If we zoom in on one of those you'll see that each SM has 4 sets of 16-wide ALUs, with each ALU capable of running a single "thread" of a shader program. Basically this thing can run *tons* of

shaders simultaneously.

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

If you've got a big hungry parallel monster, then you need to keep it fed with as many threads as possible. And this means overlapping lots of different threads, potentially from completely different draw or dispatch calls. This overlapping of draws/dispatches can occur naturally when there's no dependencies between them, and in the case of pixel shaders can actually happen because of special hardware on the GPU. The ROP units on typical

desktop GPUs are responsible for actually writing to the memory of a render target texture, and these guys can actually take out-of-order output from pixel shaders and put them back into triangle/submission order. This allows the pixel shaders to run out-of-order and overlapped, while still producing the correct results according to the API. We want all of this overlapping to occur because it makes things faster, but we’re going to get into later on.

GPU Thread Barriers

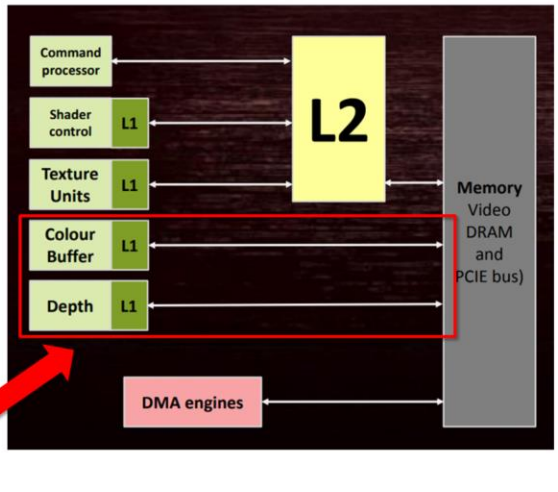
- Dependencies between draw/dispatch/copy
- Wait for batch of threads to finish
  - Same as CPU task scheduler
- Often called “flush”, “drain”, “WaitForIdle”

If the GPU is a big parallel machine that can execute tasks with dependencies between them, then we need some kind of thread barrier in order to avoid overlapping those dependent tasks. GPU operations like Draw and Dispach will generally produce large batches of threads that each process a single work item, and so whenever there’s a dependency we need some sort of mechanism that can wait for the all threads of the previous batch to run to

completion before allowing the next batch to start executing. If you've ever worked with a CPU-based task scheduler you probably have some sort of mechanism like this for resolving dependencies between tasks. On a GPU you typically have the same thing, just on a much large scale in terms of the number of threads and cores. On the GPU you might also see this kind of barrier called a "flush", "drain", or "WaitForIdle", since they essentially wait for a while bunch of threads to make their way through the entire pipeline.

GPU Cache Barriers

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

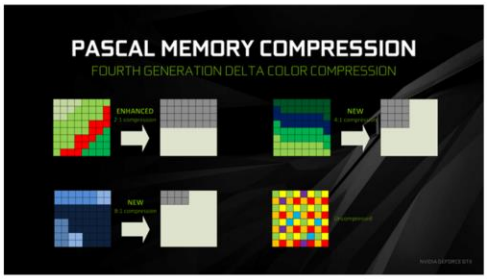


Historically, GPUs have lots of small, special purpose caches for various things. This is different from CPUs, which typically have a very straightforward hierarchy. In general GPUs are moving towards more unified cache structures, but there still are a lot of weird cache setups on video cards out in the wild. Another major difference from CPUs is that GPU caches are often not actually coherent with each other. If you look at the diagram on the right

(which is from an older AMD presentation detailing a pre-Vega GCN GPU), you can see how there are special L1 caches for memory operations involving render targets and depth buffers. These guys don’t actually go through the big L2 cache, which is a problem if we want to read a color or depth buffer as a texture in a shader. This may require flushing the contents of the color/depth L1 caches to main memory in order to make sure that the writes are visible to shader units, and the L2 may need to be invalidated in case it contains stale data due to writes sitting in the color/depth L1. This is one of several reasons why people always tell you to batch your barriers: flushing a cache once is always going to be faster than flushing it multiple times in a row.

GPU Compression Barriers

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



A third type of barrier deals with compression and layout changes. GPUs typically have special hardware that allows them to enable lossless compression when writing to render targets or depth buffers, which is typically done to reduce the internal bandwidth required. Nvidia and AMD both employ forms of Delta Color Correction for render targets, which generally work by taking advantage of coherency within a block of pixels. In other cases there can also be

special metadata buffers that can indicate if a tile of pixels is in a “clear” state, which saves having to write the actual clear value to the texture prior to rendering. In both cases it may be required to perform decompression operation to write the expanded values to all texels before the render target or depth buffer can be read as a texture, which is something that would happen when a barrier indicates that the resource is going to be read from. It’s also possible that the hardware may not be capable of reading or writing to a compressed resource as a UAV, would may require a mid-frame decompression.

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

When we work with D3D12, we don’t actually work with low-level barriers. It’s not like you issue commands to flush caches, or decompress buffers. That wouldn’t really make sense for an API that’s meant to work with all kinds of hardware, since those low-level operations reflect hardware-specific details. Instead you work with a higher-level abstraction where the barriers reflect transitions from one state to another. These generally describe the visibility of a resource to

a particular part of the logical D3D12 pipeline: for instance you may say that a texture is going to be read by a pixel shader as an SRV, or that buffer is going to be written to as a UAV. These transitions implicitly represent dependencies that the driver can infer, and then convert into the necessary low-level barriers.

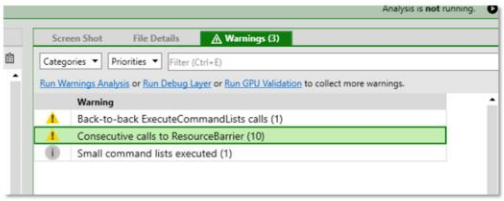
Vulkan Barriers

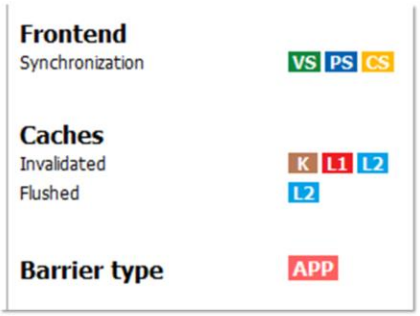
- More explicit (verbose) than D3D12
- Specifies
  - Producing/consuming GPU stage
  - Read/write state
  - Texture layout

Vulkan works at a similar level with its pipeline barriers, which also abstract away the hardware-specific details. However they are a bit more verbose than D3D12, requiring you to explicitly specify the producing and consuming GPU stages, the read or write state, and the 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!





Either way, both D3D12 and Vulkan are abstracting things away from you. But with that said, the drivers are still going to be issuing their low-level barriers as a direct response to the manual barriers that your app is submitting. So you're still firmly in the driver's seat here. This means that you can absolutely issue too more barriers than necessary resulting in unnecessary syncing, flushing, or decompressing. The good news is that the Radeon Graphics

Profiler will show you what’s going on in response to one of your barriers, as seen in the image on the right. It will show you what kind of thread synchronization operations are occurring, as well as which caches are being flushed and/or invalidated. It will even show you whether the barrier is happening in response to an API barrier, or if its something it had to do implicitly as the result of other API calls. If you’re interested in performance, you should also take a capture with PIX for Windows and look at the warnings tab. It will tell you whenever you’re failing to batch your barriers, and are instead calling ResourceBarrier multiple times in a row with no draw/dispatch/copy calls in between.

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!

You may be wondering why we didn't need to deal with any of this barrier stuff back in D3D11. The reason why is because the driver was actually putting them in for you. So there were still syncs and flushes behind the scenes, you just weren't responsible for manually inserting them. To do this though the driver has to analyze your commands as you submit them to look for dependencies, and then insert in the appropriate barriers as necessary. In

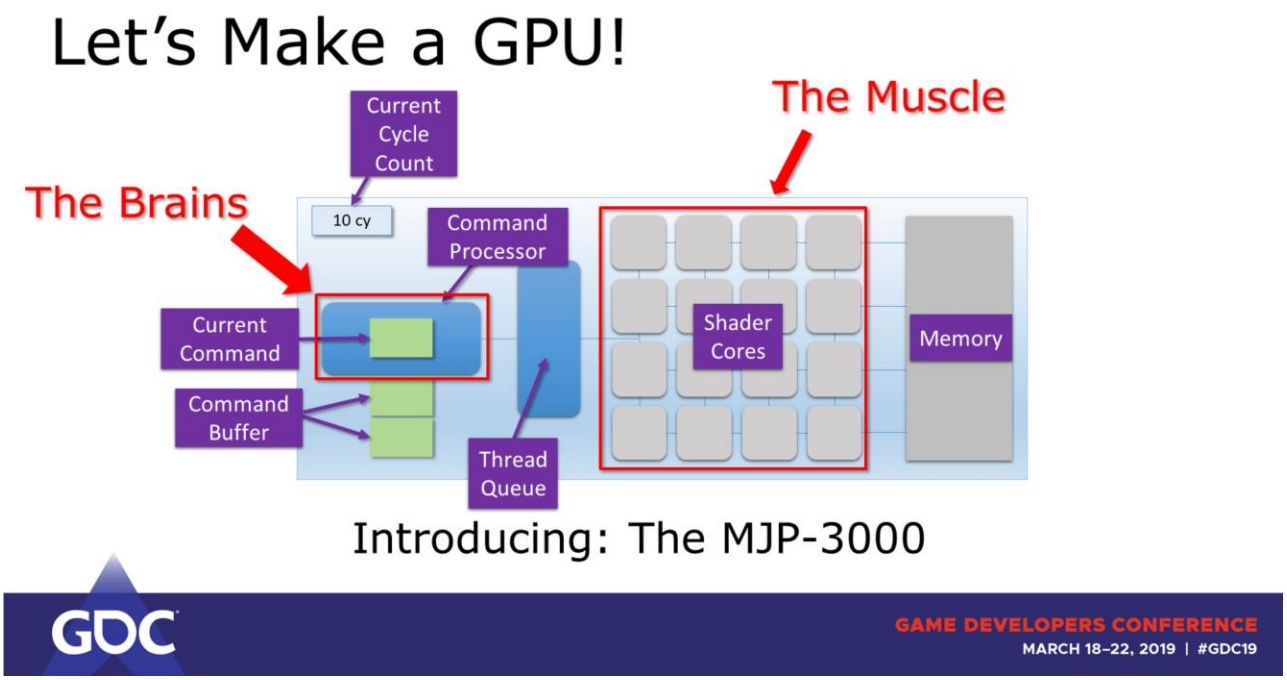
that way it was like a run-time compiler, analyzing an entire frame's worth of commands on-the-fly. Basically it was easy mode, just like single-threaded CPU programming.

So why did we give this up if it was so nice to work with? For starters, it requires a ton of CPU work to figure out all of these dependencies automatically. Everyone wants their games to run fast, and doing this analysis can add significant overhead to rendering. On top of that, this kind of automatic analysis doesn't play very well with multi-threaded command submission. This is because dependencies can span the chunks submitted different threads, which may require the driver to defer final command buffer generation until it receives and stiches together all of your commands from all threads. Finally, you really need to have all of your resource bindings be visible to the driver if its

going to sort out the dependencies for you. The driver can’t know that a draw is going to depend on an earlier dispatch unless the driver can see all of the resources that the draw will read from, which essentially forces into you having some kind of CPU-based binding API like you had in D3D11. This means no GPU-side mechanisms for binding textures, and no bindless-style rendering where all resources are globally visible to a shader. These three things are fundamentally incompatible with the D3D12/Vulkan worldview, which are all about enabling the lowest-possible CPU overhead and supporting multi-threaded command buffer generation.

Side note not mentioned in the presentation: in some cases the barriers were actually implied by the way the D3D11 spec was written. In particular, the spec required that drivers always insert a barrier between draws or

dispatches that used a UAV in order to ensure that writes were visible. This is relaxed in D3D12, since you can now manually issue UAV barriers as needed. Nvidia and AMD both actually have extension APIs that can let you disable the auto-barrier after a UAV write, which can let you get better performance.



With all of that intro material out of the way, I'd like to dive into some more specifics about thread synchronization on GPUs. To do that, I thought it would be more useful if we looked at a simplified example GPU instead of complex real-world example. That way it ill be easier to understand what's going on, while also not being misleading about how an actual piece of hardware works. So on that note I would like to present to you: The MJP-3000! This

beautiful modern marvel of engineering was designed by yours truly, so feel free to reach out after the talk if you think you'd like to license this at AMD or Nvidia. ;)

On the left side you'll see a part labeled the command processor, which is a unit that process commands from a command buffer one at a time. These commands will result in batches of threads getting deposited in the thread queue immediately to its right. Waiting threads in the thread queue will get pulled out by the shader cores, which is where the shader programs will actually execute and do their work. These programs can then read and write to the memory on the far right side. On the top left there's also a cycle counter, which will tell you how much time has elapsed when we go through some examples using this GPU.

Broadly-speaking, I would label the command processor as “The Brains”, since it executes the commands and decides which work will actually be done. Meanwhile I would call the shader cores “The Muscle”, since they’re dumb (they can’t tell themselves what to do) but they do all of the heavy-lifting.

You may also see me refer to the command processor + thread queue as a “Front-End”, since this is the portion that feeds threads into the shader cores.

MJP-3000 Limitations

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

This GPU I've designed is considerably simplified compared to a real-world GPU. For starters, this thing can only do compute operations (dispatches). Handling the full rasterization pipeline would add considerable complexity, and I'd like to avoid that. It's also only got 16 shader cores, which is significantly fewer than your average desktop GPU. There's also no SIMD for those shader cores, which is commonly used on GPUs to improve throughput.

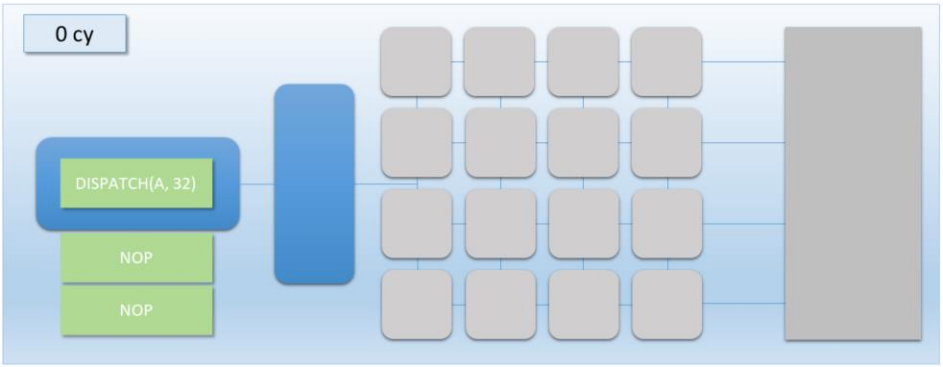
What this means for our examples is that each shader core is considered completely independent, and each can be running a totally separate shader program. On a related note there's also no thread cycling, which is a technique used on GPUs to hide latency of long operations by switching to another waiting thread. So once a thread starts running on one of our shader cores, it will run to completion. Finally, there's no caches on this thing. We're not going to be looking at flushing incoherent caches or anything like that, I mainly just want to talk about thread synchronization.

Simple Dispatch Example

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

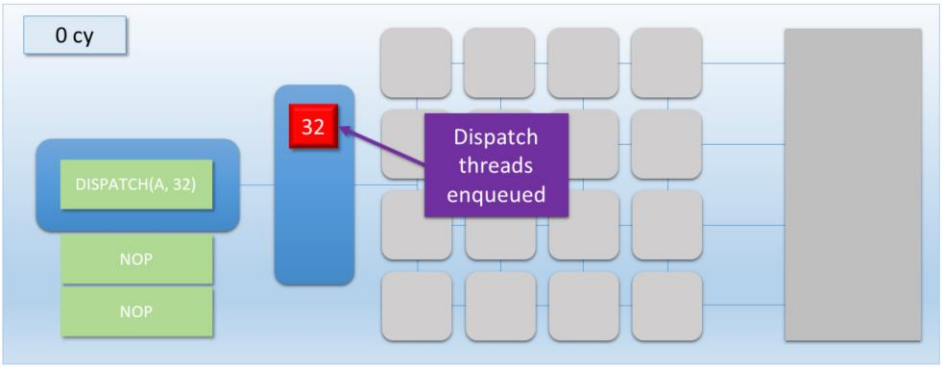
Alright, let’s try going through a simple example on our GPU. Let’s say we want to dispatch 3D threads of a shader program that we’ll call Program A, and each thread of that program will write 1 element to memory. We’ll also say that this program takes 100 cycles to completely finish.

Simple Dispatch Example



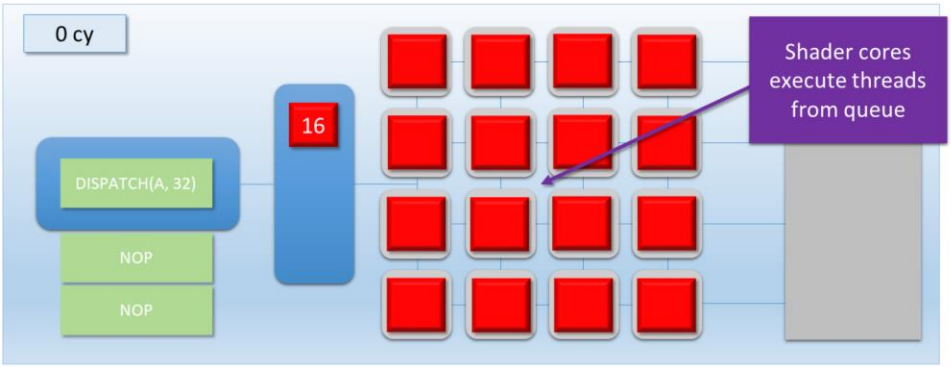
We start out at cycle 0 with a DISPATCH command ready to be executed by our command processor.

Simple Dispatch Example



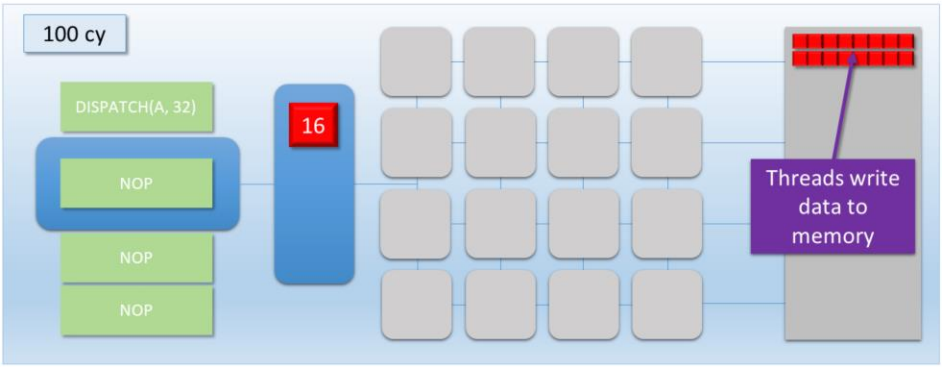
The command processor immediately deposits 32 threads of program A in the thread queue

Simple Dispatch Example



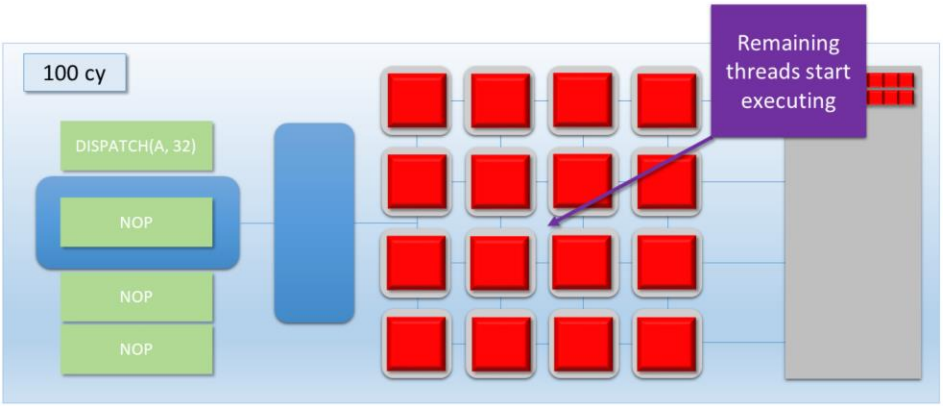
The threads in the thread queue then get pulled onto the shader cores, where they actually start executing.

Simple Dispatch Example



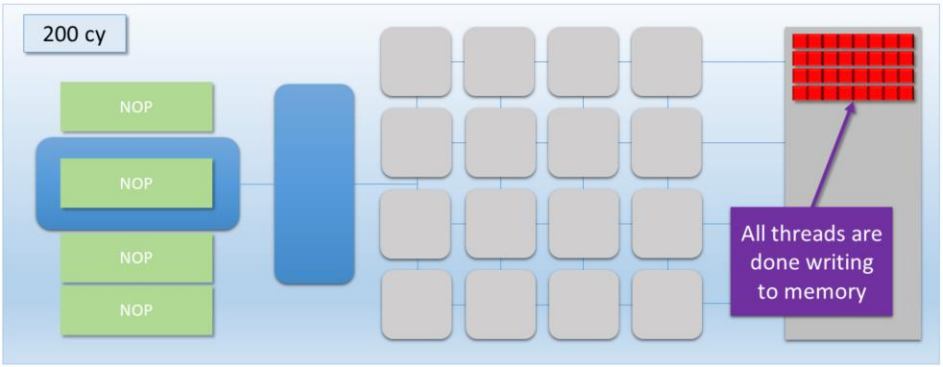
At cycle 100 the first batch of threads have completed their program, and the results have been written to memory.

Simple Dispatch Example



At this point the remaining 16 threads in the thread queue can have their turn on the shader cores

Simple Dispatch Example



The second batch takes another 100 cycles to finish, so we’re completely finished at cycle 200.

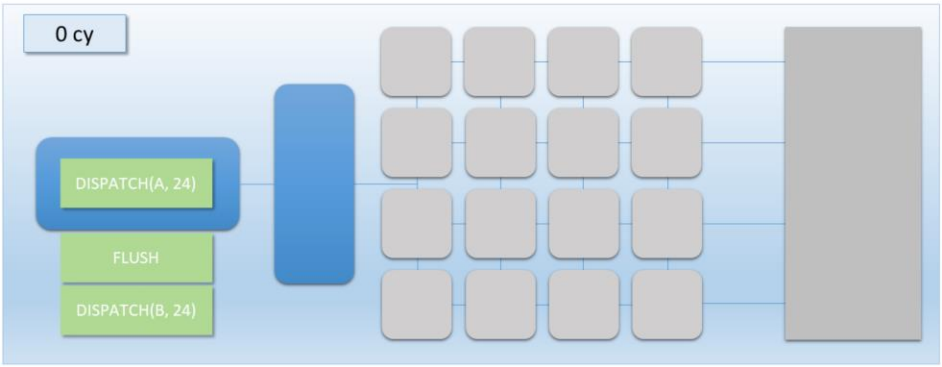
Thread Barrier Example

- 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

Now that we see how things work, let's try a more complicated example. Let's say we now have two dispatches, which we'll call A and B. Threads from dispatch A (colored red) will run first to produce some data, and that data will be read by the reads from dispatch B (colored green). This means there's a dependency between these two dispatches, which also means that can't overlap at all! To make sure that all of the data is visible to

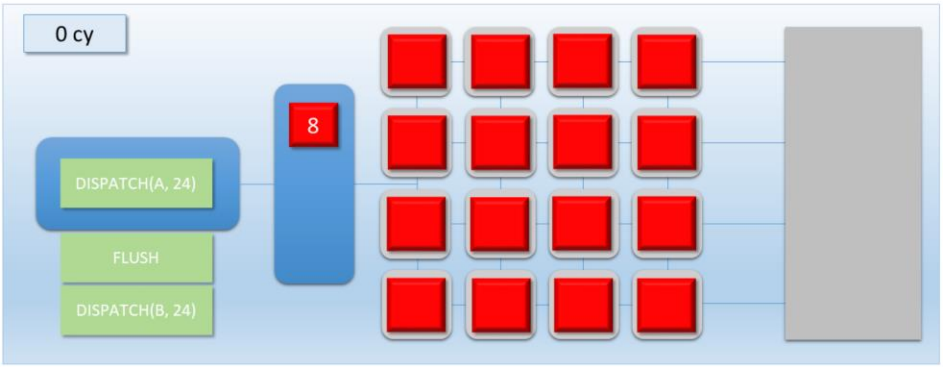
dispatch B, we need to introduce a new command called FLUSH. When the command processor hits this command, it will sit there and wait for the thread queue and shader cores to become completely empty before it moves on to the next command. This will let us avoid the overlap in this case.

Thread Barrier Example



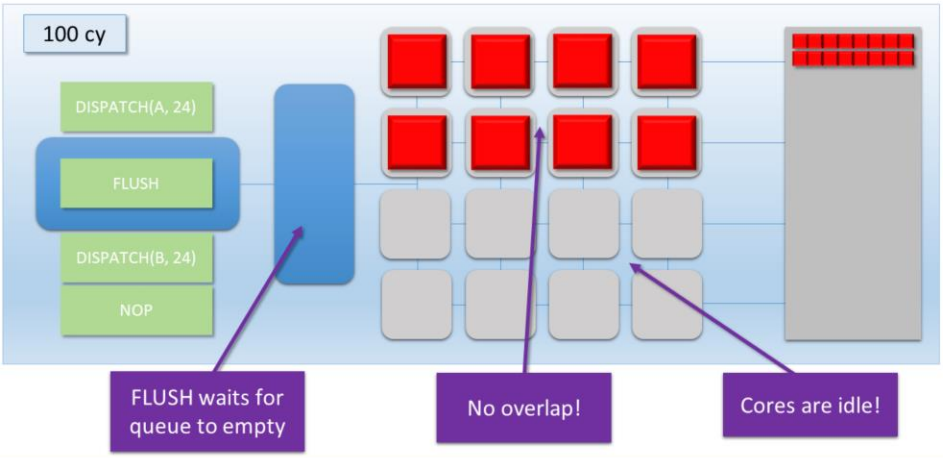
Let’s see how this actually plays out. We start again with the DISPATCH A command, this time with 24 threads.

Thread Barrier Example



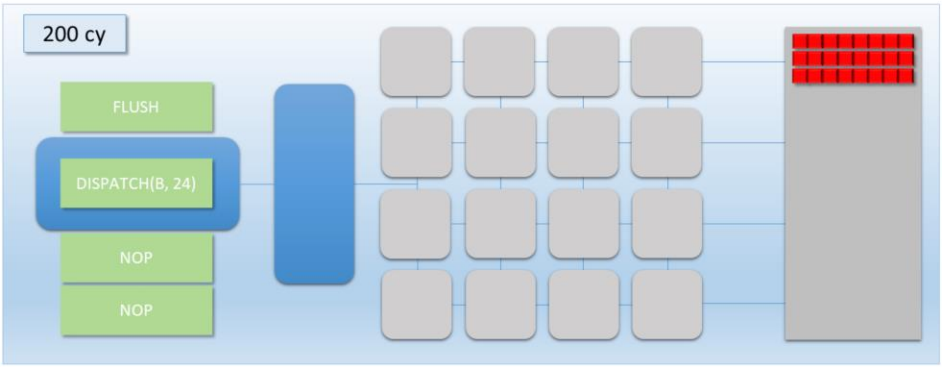
This leads to the first batch of 16 threads of program A to start running on the shader cores

Thread Barrier Example



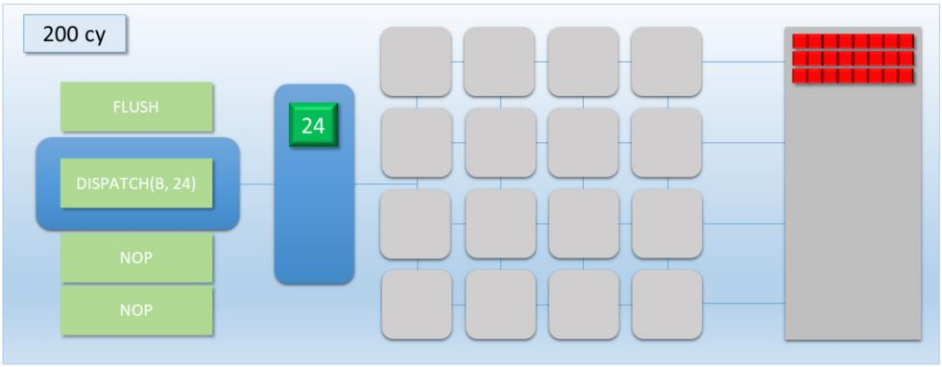
If we jump to 100 cycles later, we see that the first batch of threads has completed. We also see that the command processor is sitting on the FLUSH command. This means the command processor will wait until all of the threads from Dispatch A to finish before moving on, which is why there's no overlap here between A and B. However this also means that 8 of our shader cores are sitting idle, since there's not enough threads left from Dispatch A to fill them up.

Thread Barrier Example



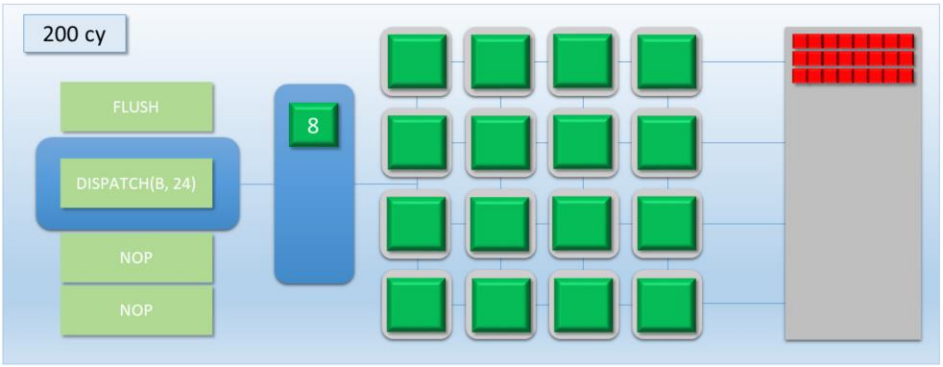
At cycle 200 all of the threads from DISPATCH a have finished, so the command processor can finally move on to Dispatch B.

Thread Barrier Example



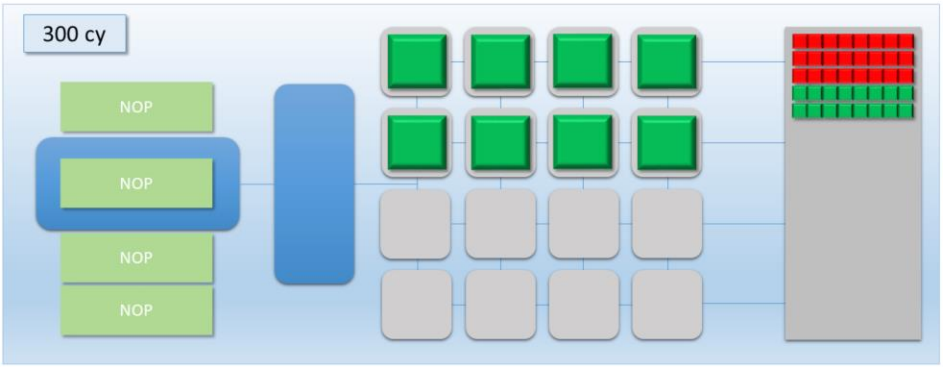
The 24 threads from Dispatch B get put in the thread queue…

Thread Barrier Example



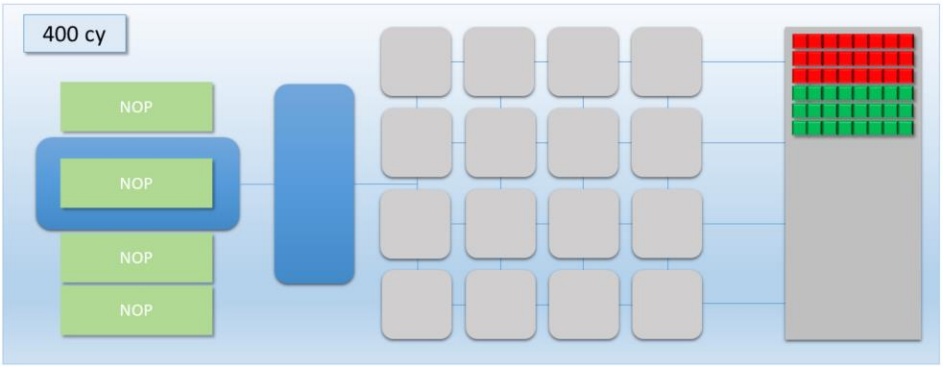
…and then the first batch of 16 threads start running on the shader cores.

Thread Barrier Example



100 cycles later the first patch of threads from Dispatch B have finished, and the second batch can start. Once again half of our cores are idle.

Thread Barrier Example



At cycle 400 we’re totally finished and everything is written to memory.

Thread Barrier Example

- FLUSH prevented overlap ☺
- …but cores were 50% idle for 200 cycles
  - 75% overall utilization ☹
  - Took 400 cycles instead of 300 cycles

So how did it go? The good news is that the FLUSH worked for preventing overlap, and we avoided any race conditions. But the bad news is that our shader cores were 50% idle for 200 cycles, leading to a 75% overall utilization rate for the full 400 cycle period. With peak utilization it would have only taken 300 cycles to execute all of those threads, so we basically have 100 cycles of overhead from the FLUSH.

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

This leads us to start to reason about how much it costs to insert a thread barrier like this on a GPU. One intuition we can glean from this is that the cost of a barrier is going to tied to the drop in utilization that's caused by the barrier. We can also flip that around and realize that the potential game from *removing* a barrier is going to be relative to the percentage of shader cores that we were leaving idle.

From this we can also reason that dispatches with lots of threads are naturally going to lead to better overall utilization, since they’re going to be able to saturate all of the shader cores on their own with tons of overlapping threads. We can also reason that longer running threads are going to lead to a very high flush cost, since whatever cores they don’t fill up towards the end are going to remain idle for a long period of time.

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!

Whenever you're issuing a barrier between draws or dispatches in D3D12 or Vulkan, you should fully expect something equivalent to a FLUSH to occur in order to resolve the dependency. The same goes for if you use a UAV barrier to force UAV writes to be visible to subsequent draws or dispatches. To improve utilization you should try to group together non-dependent draws and dispatches between your barriers so that the GPU has more to chew on

before syncing.

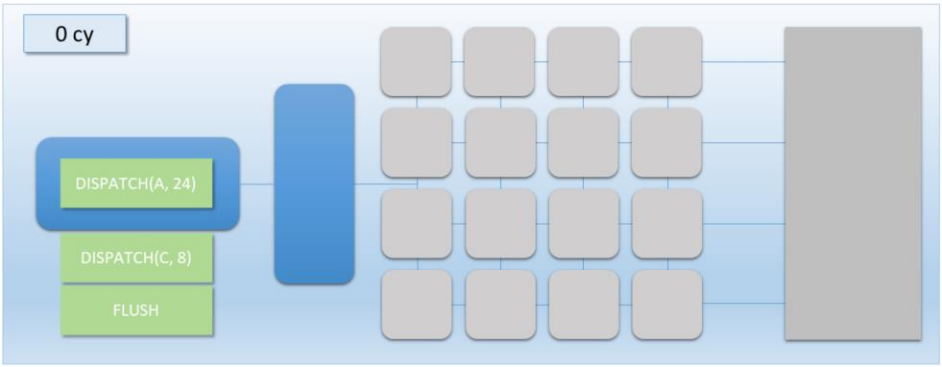
One thing to keep in mind though is that I’m really only talking about currently-available GPUs that are on the market right now. It’s totally possible that future hardware will do things differently, and won’t require the same kind of low-level barriers that we talked about earlier.

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

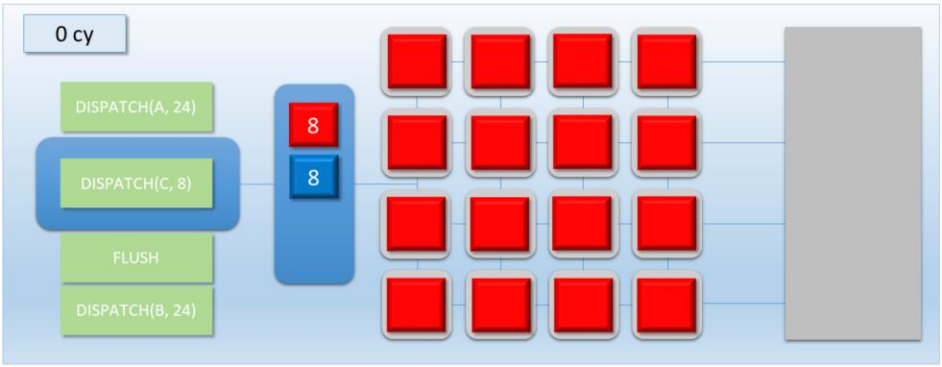
Let’s see if we can improve our utilization by overlapping the Dispatch A with some threads from another dispatch, which we’ll call Dispatch C (colored in blue). We can do this because Dispatch C isn’t dependent on either A or B, which means it’s safe to overlap with either of them.

Overlapping Dispatches Example



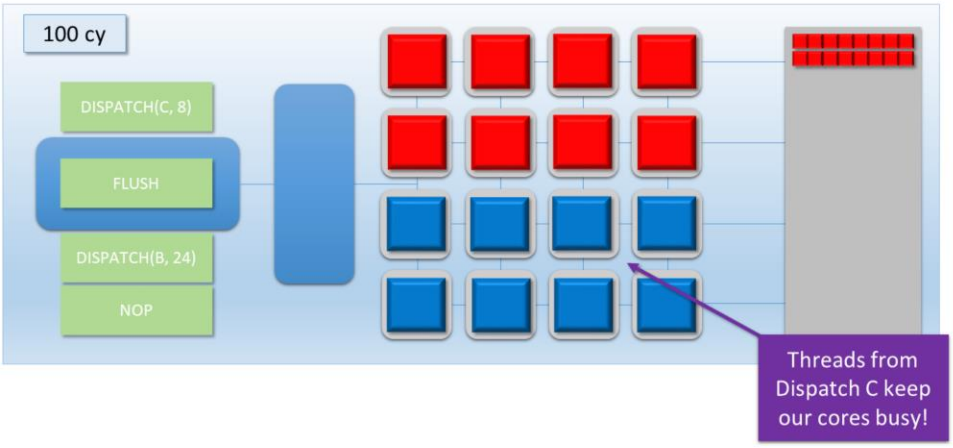
Once again we start out with Dispatch A, which is followed immediately by Dispatch C in the command buffer.

Overlapping Dispatches Example



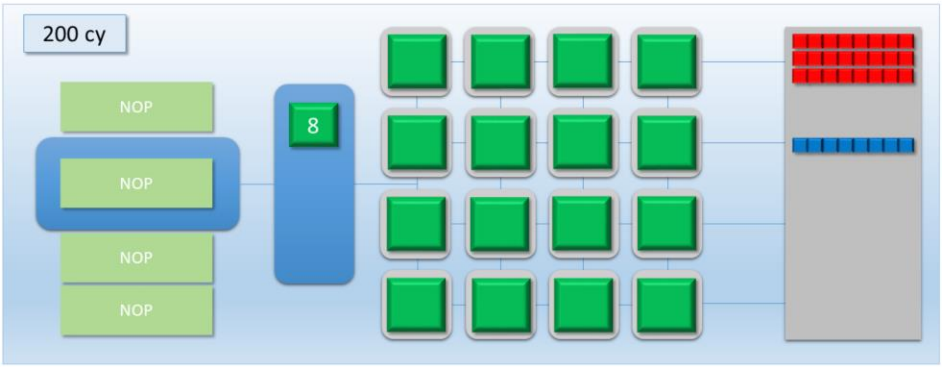
This causes all of the threads from A and C to get deposited in the thread queue. Dispatch A's threads get on the shader core first, since that command was processed first by the command processor.

Overlapping Dispatches Example



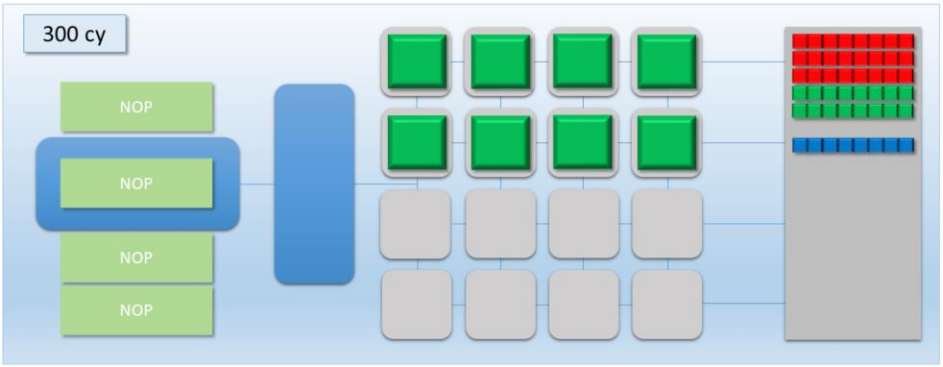
At cycle 100 we see that the command processor is once again sitting on the FLUSH, waiting for all of the threads to finish. But this time we have full utilization for this period, because the 8 threads from Dispatch C were able to take up the remaining cores that were idle last time around.

Overlapping Dispatches Example



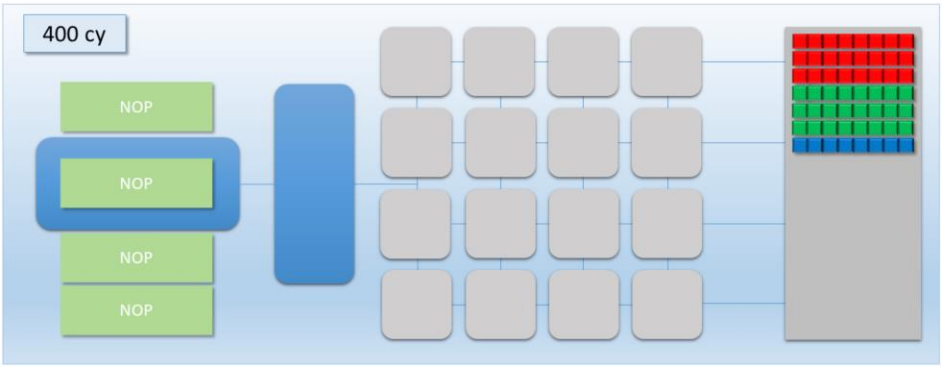
At cycle 200 both A and C have finished, so Dispatch B can start doing its thing.

Overlapping Dispatches Example



If we jump 100 cycles later the first batch from Dispatch B has finished…

Overlapping Dispatches Example



…and then at cycle 400 we're completely finished.

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!

So how did we do this time? You might notice that the whole sequence still took 400 cycles, which means we had the same end-to-end latency for Dispatch A + Dispatch B. However we were basically able to sneak in Dispatch C for free with no extra processing time. This means that our overall throughput has increased for the whole sequence. If we were to submit A + B + C sequentially with barriers in between each of them it would have taken 500 cycles to

complete, so we saved 100 cycles with this approach. Our utilization also jumps up to 87.5%, which is great!

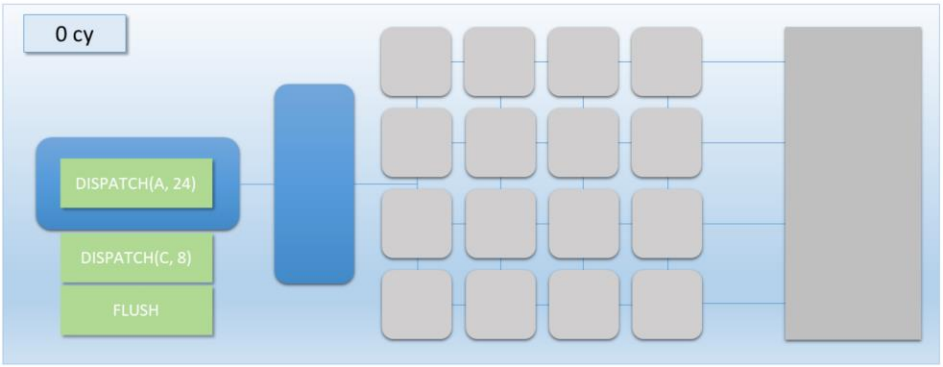
Insights From Overlapping

- 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)

Before moving on, what if we were to take our glasses off and squint a bit while looking at these examples? If we do this, our GPU basically looks roughly like a CPU, where we process individual commands instead of instructions. From this point of view, overlapping our dispatches is basically like achieving Instruction Level Parallelism on a CPU. The catch is that this parallelism is explicit in our case, since we have to carefully order our commands in such a way

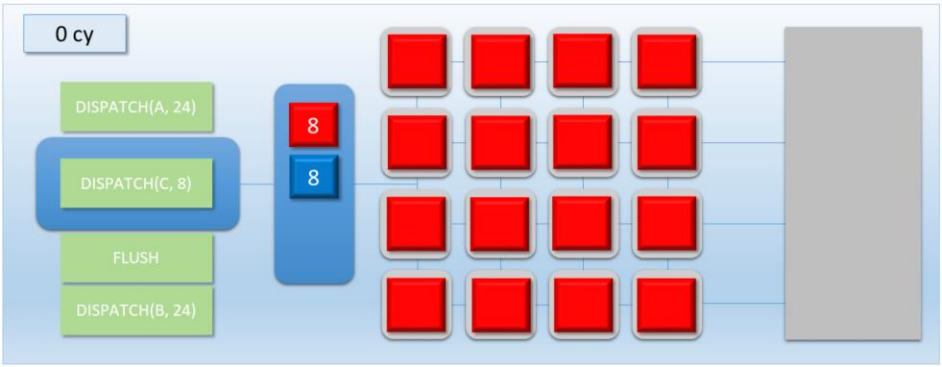
that they overlap while also using barriers to synchronize. In that way it’s probably more like Very Long Instruction Word architectures, which require the compiler to explicitly identify and schedule parallel operations

Bad Overlap Example



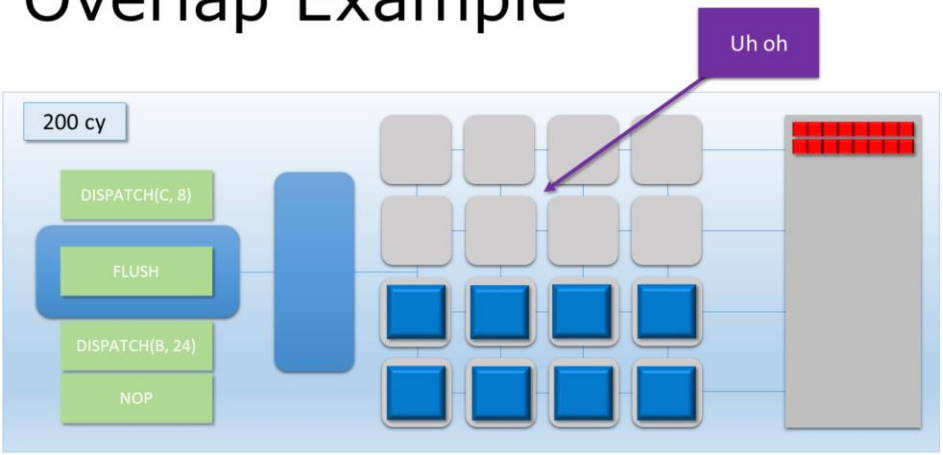
So with the previous example things worked out great because Dispatch C was the perfect size and length to exactly fill in the idle cores during the FLUSH. This is obviously fairly contrived, and you can easily imagine scenarios where this wouldn’t work out so nicely. Let’s try the previous example again, except this time the program for Dispatch C takes 400 cycles to execute instead of 100. We’ll start out again with DISPATCH A…

Bad Overlap Example



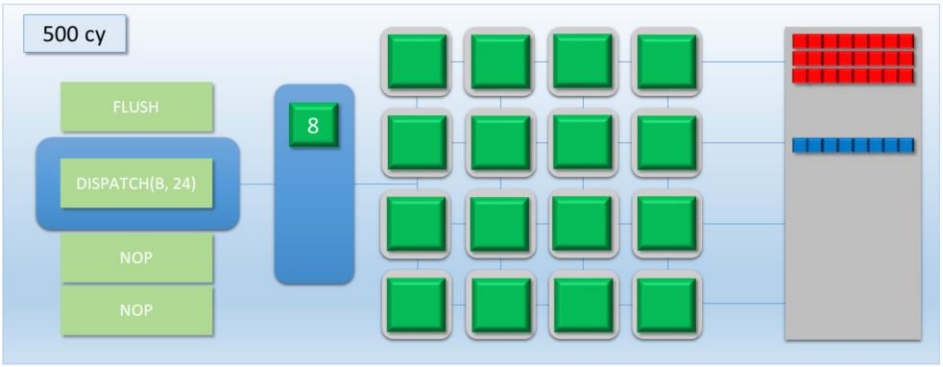
…and again all of threads from A and B get queued up to run.

Bad Overlap Example



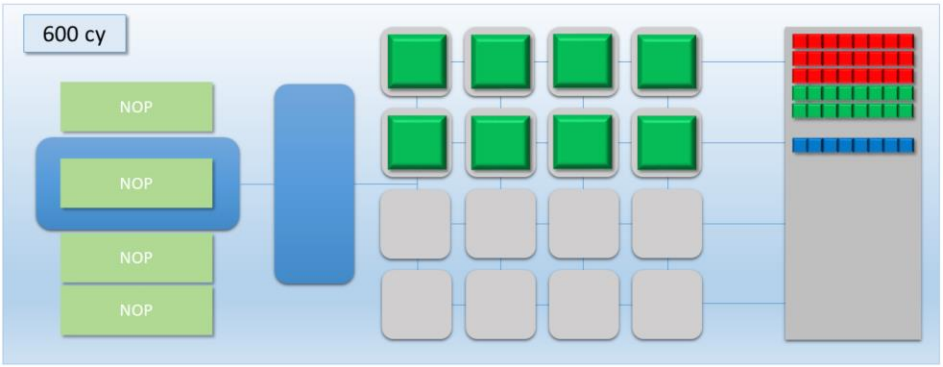
If we jump 200 cycles into the future we see that something unfortunate has happened: Dispatch A is finished, but the FLUSH is causing the command processor to just sit around and wait for Dispatch C to finish. This is pretty bad since Dispatch C still has another 300 cycles to go, and we're only hitting 50% utilization.

Bad Overlap Example



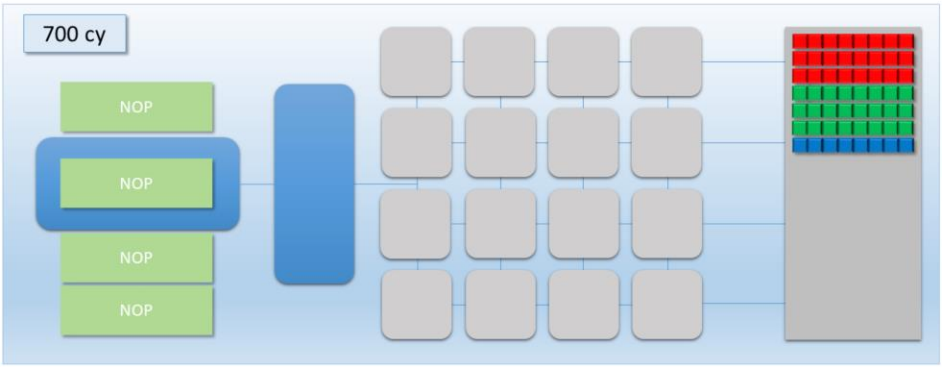
We have to jump all the way to cycle 500 before Dispatch C is finally finished, and the command processor can move on to Dispatch B.

Bad Overlap Example



100 cycles later the first batch of Dispatch B is finished, and we're again at 50% utilization.

Bad Overlap Example



Then at cycle 700 we’re finally(!) finished.

What Happened?

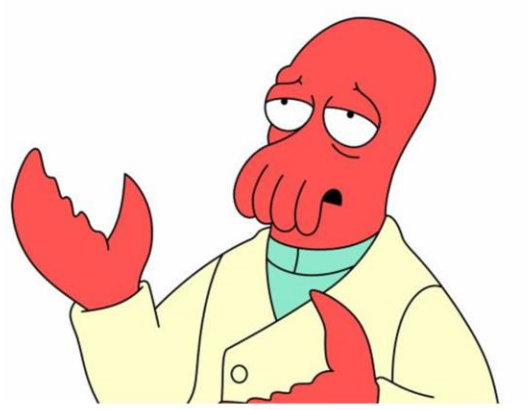
- 400 cycles with 50% idle cores
  - 71.4% utilization
- 1 CP -> 1 queue -> global flush/sync
  - B wanted to sync on A, but also synced on C
- Re-arranging could help a bit
  - But wouldn't fix the issue

Yikes! That really did not work out for us this time. We had a total of 400 cycles where we were only using half the cores, which drops our overall utilization to 71.4%!

The problem we had here was caused by the fact that we only have one front-end here: Dispatch B really just wanted to sync on Dispatch A, but it ended up syncing on Dispatch C as well. Basically a FLUSH is global sync operation, which turns it into a very

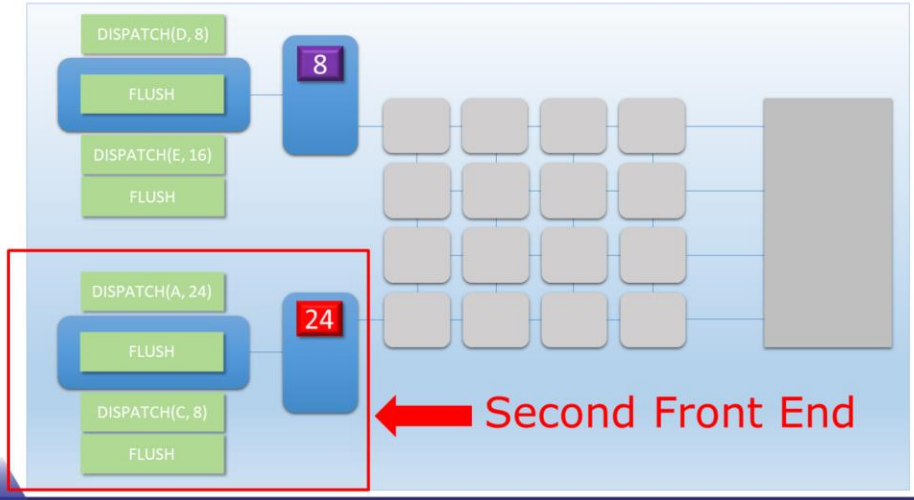
blunt instrument. In some cases with a similarly long dispatch we could potentially improve things a bit by re-arranging our commands, but in general this is hard to do effectively because we typically don’t know exactly how long a thread will take on a real GPU.

Why Not *Two* Command Processors?



If our problem in that last example was caused by only having one command processor, you might be thinking to yourself “why not just stick another command processor on there?”

Upgrading To The MJP-4000



If you were thinking that, it turns out that the engineers at MJP Industries were way ahead of you. For the next revision of the MJP series they decided to go ahead and stick second front-end in there that feeds into the shader cores. Unfortunately this caused them to blow through their transistor budget, which left no room left on the die for additional shader cores. So we're still stuck at 16 here.

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

With two front-ends we can effectively process to command streams at once. This is going to work out best if these two streams don't depend on each other in anyway, so we really want two *independent* command streams to properly feed this thing. We also have to be careful since our max throughput hasn't increased relative to the MJP-3000, since we still only have 16 shader cores. So we're not going to do any better for the parts

that were already running at 100% utilization.

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

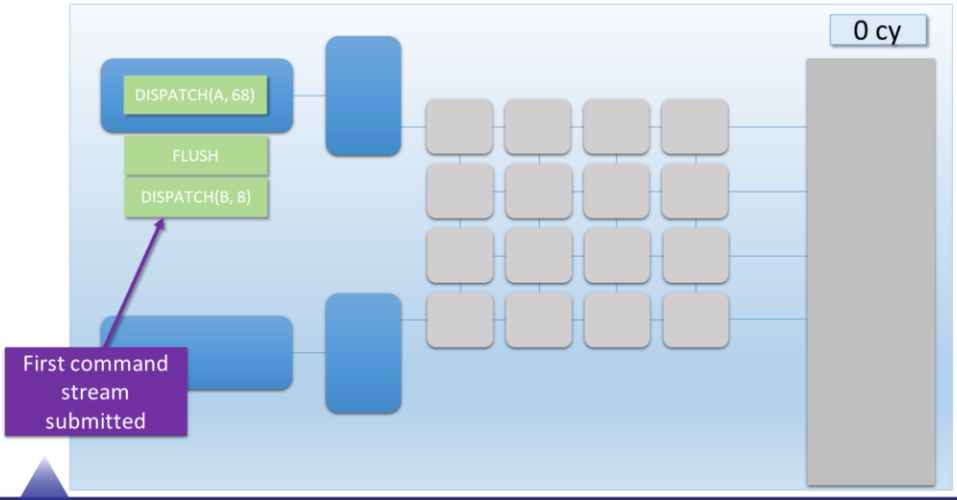
Let’s now go through an example that utilizes both front-ends. We’ll start out with Dispatch A, which this time has 68 threads that take 100 cycles to finish. We’ll follow that up with Dispatch B, which only has 8 thread that take 400 cycles to finish. We’ll say that B depends on A, which means we’ll have to have a FLUSH between them.

Meanwhile we’ll also have Dispatch C,

which has 80 threads that take 100 cycles to complete. That will be followed by Dispatch D, which also has 80 threads that take 100 cycles to finish. Dispatch D depends on C, so this will also need a FLUSH.

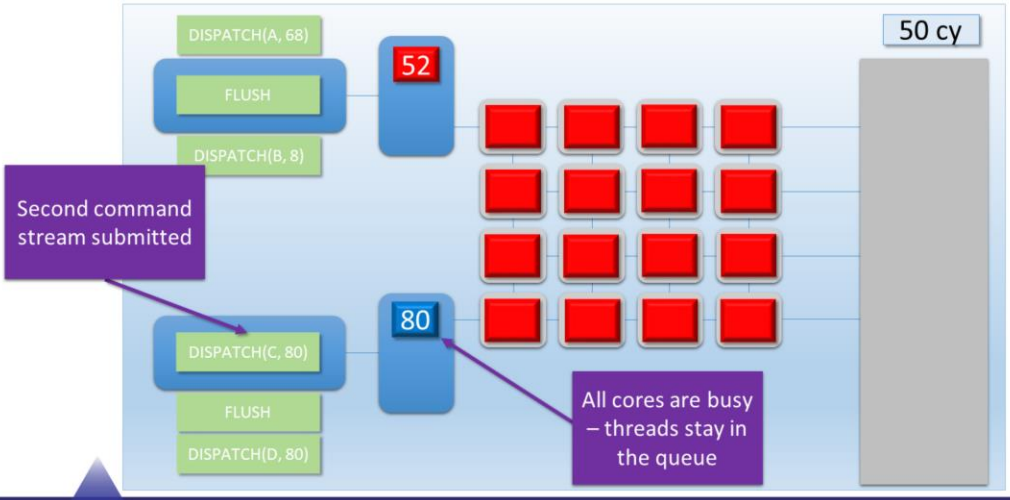
We can see here that we have two independent command streams, which is exactly what we want for the MJP-4000.

Dual Command Stream Example



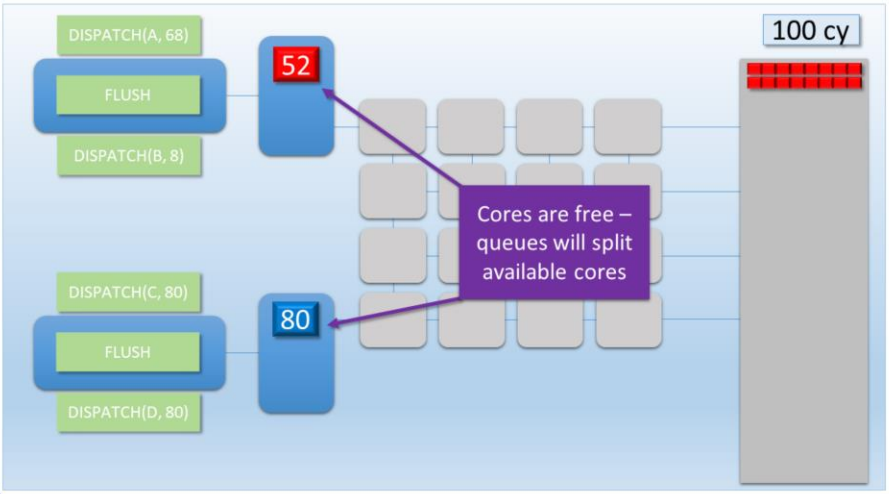
We start out by submitting the first command stream to the top front-end.

Dual Command Stream Example



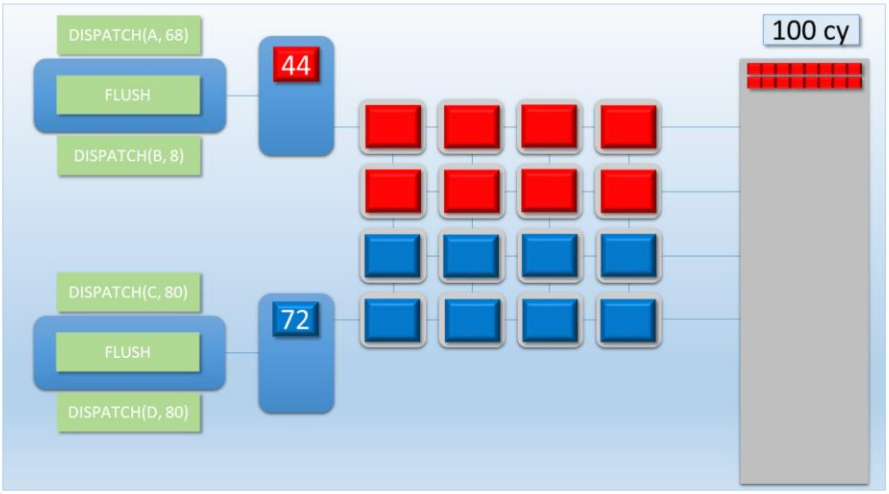
After 50 cycles, the first batch of Dispatch A's threads are running on the shader cores, and the remaining 52 threads are in the top thread queue. At this point we submit the second command stream to the bottom front-end. All 80 threads from Dispatch C end up in the thread queue, since the shader cores are all busy at the moment.

Dual Command Stream Example



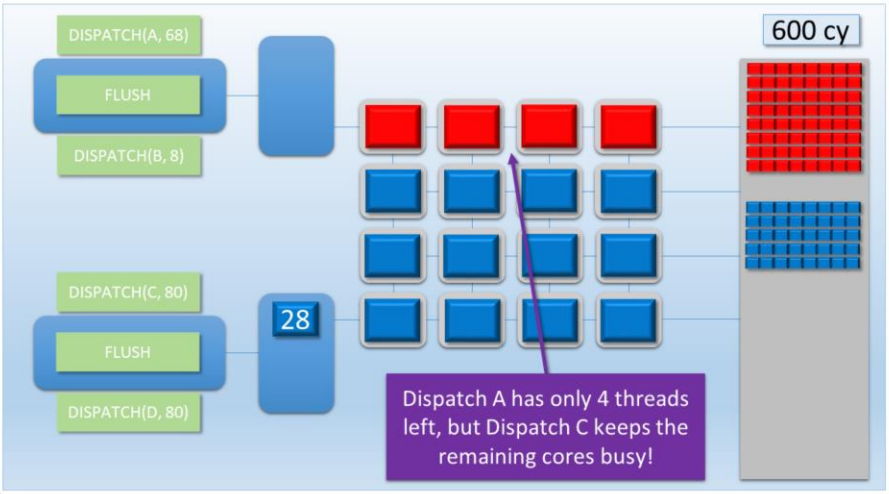
At cycle 100 the first batch of Dispatch A's threads have finished, so the shader cores can pull evenly from both thread queues.

Dual Command Stream Example



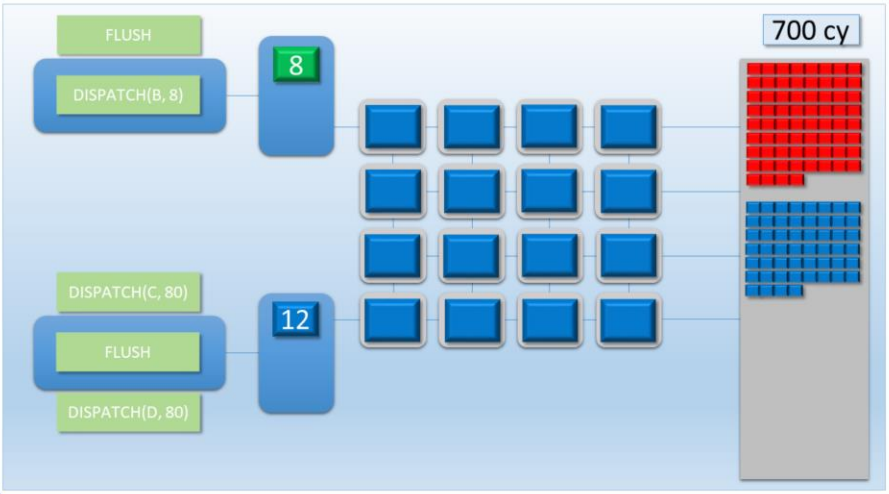
Dispatch A now has 8 threads running, and Dispatch C uses the remaining 8 cores. Both command processors are also sitting on a flush, waiting for their respective dispatches to finish.

Dual Command Stream Example



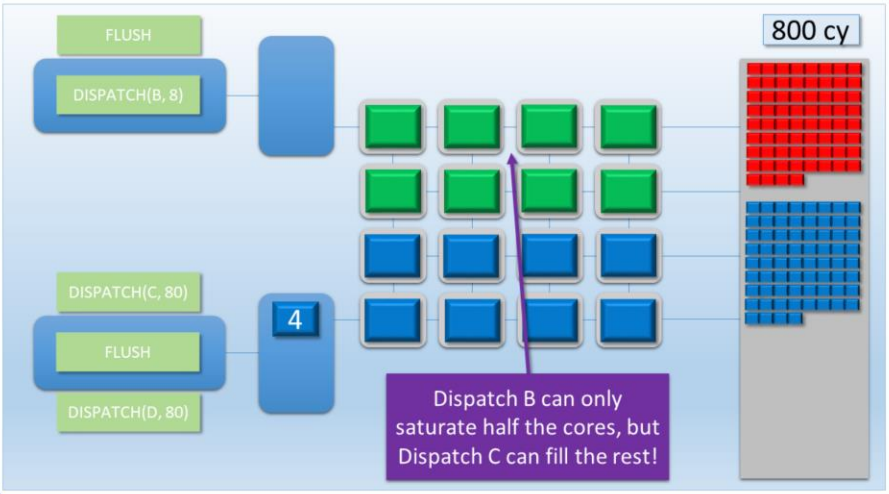
We now jump way ahead to cycle 600, where Dispatch A is winding down. At this point there are only 4 threads remaining from Dispatch A, which would normally mean that we would have 100 cycles with only 25% utilization. However we have plenty of threads left from Dispatch C to fill those remaining cores, which lets us stay at full utilization.

Dual Command Stream Example



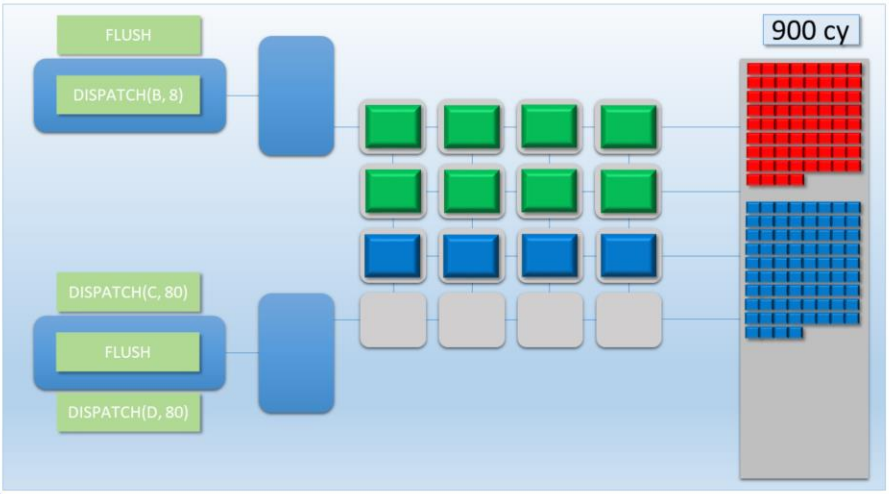
After another 100 cycles Dispatch A is finally finished, so the top command processor can move on from the flush to Dispatch B. Those 8 threads will have to wait another 100 cycles to actually run though, since all of the shader cores are busy at the moment with threads from Dispatch C.

Dual Command Stream Example



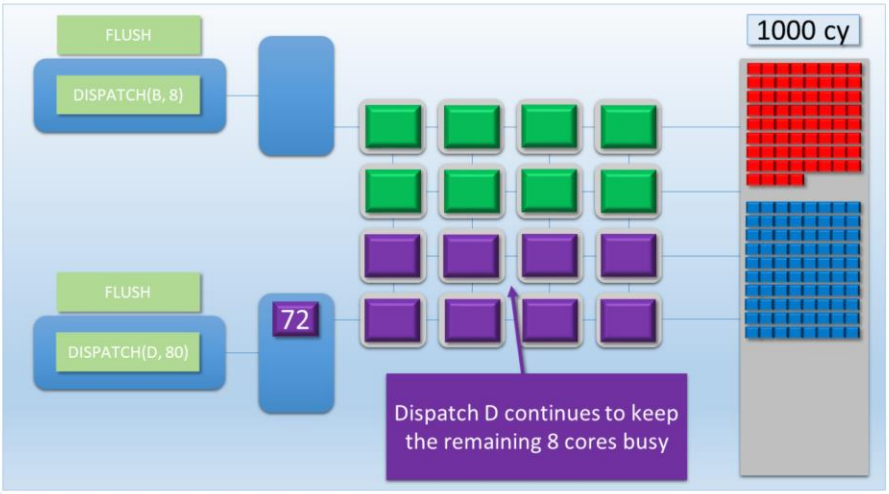
After another 100 cycles the current batch of Dispatch C threads complete, which means B and C can both split the cores. It’s worth noting that the 8 threads from Dispatch B can only fill half the cores, which again would be really bad if we didn’t have Dispatch C in the mix!

Dual Command Stream Example



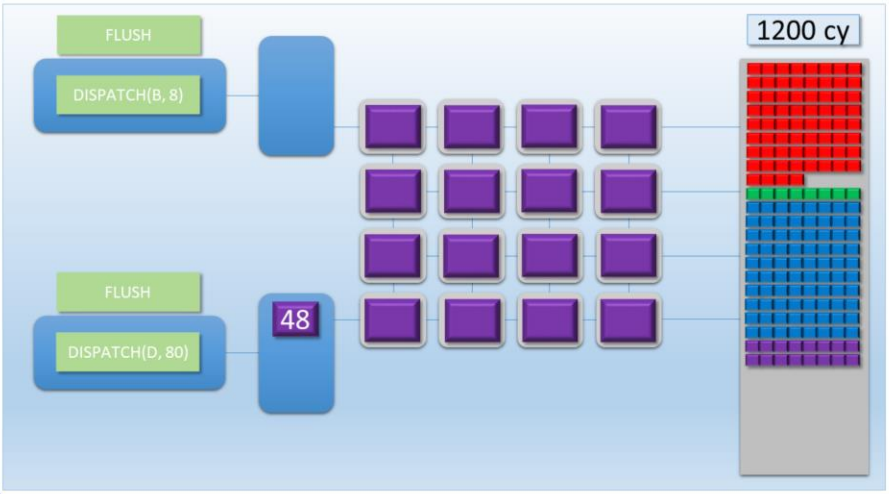
At cycle 900 Dispatch C is winding down, giving us a brief period with only 75% utilization.

Dual Command Stream Example



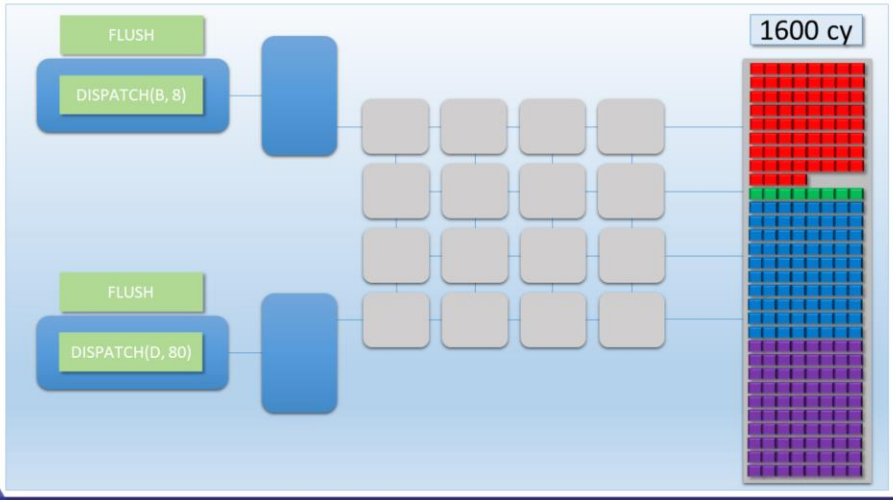
After another 100 cycles the flush on the bottom command processor can finally finish, allowing it to move on to Dispatch D. This lets the threads from Dispatch D fill up the remaining 8 cores that Dispatch B isn’t using, once again bringing us to 100% utilization.

Dual Command Stream Example



After another 200 cycles Dispatch B has finished, letting Dispatch D have all of the cores to itself.

Dual Command Stream Example



The whole sequence finally finishes after about 1600 cycles. Whew!

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

So did having two front-ends actually help us in this case? I would say that it absolutely did! We averaged 98% utilization for the whole 1600-cycle sequence, which is fantastic considering we had Dispatch B in there that only had 8 threads and took 400 cycles to finish. If we were to do the naïve version where we submitted all 4 dispatches sequentially on the same front-end with barriers in between, the whole sequence would have taken 1900

cycles to complete and would have achieved only 85% utilization. The one catch here is that the end-to-end latency actually *increased* vs. the sequential version if you were to look at A + B or C + D as individual sequences. This is because the two command streams were sharing the shader cores, causing the dispatches to take longer to fully execute. This can be a concern if you really want low latency for a dispatch, but if you only care about overall frame time then it’s definitely a win.

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

On real-world GPUs this sort of "multi-command-stream" submission can potentially work out even better in some cases. Real shader workloads will often have long stalls where they wait hundreds or thousands of cycles for memory operations to complete, which GPUs like to hide by swapping out the running thread to a different waiting thread (I called this "thread cycling" earlier). Submitting more work from a different front-end can potentially give the GPU more

threads to switch to, giving it more ability to hide latency. If you recall the earlier slides, you'll remember we also mentioned that cache actions like flushes and invalidations are typically a part of resolving dependencies. If one command processor is sitting around waiting for a cache flush, that's a good opportunity for another command processor to take advantage of those idle cores by launching more threads. There's also certain workloads that naturally lead to low shader core utilization just due to their very nature. The most common of these is depth-only rasterization, which tends to be bottlenecked by fixed-function rasterization and depth-buffer hardware while only launching very minimal amounts of vertex shader threads. Certain implementations of tessellation and geometry shaders also try to keep all of the work on the same group of shader cores in order to avoid spilling to memory, which can lead to choke points where other groups of

shader cores go unused. Finally, GPUs also tend to have fixed-function DMA units that can handle memory transfer operations without using any of the shader cores. If a command processor is waiting for a DMA to finish, that’s another opportunity for some non-dependent work to sneak onto the idle cores and get some work done.

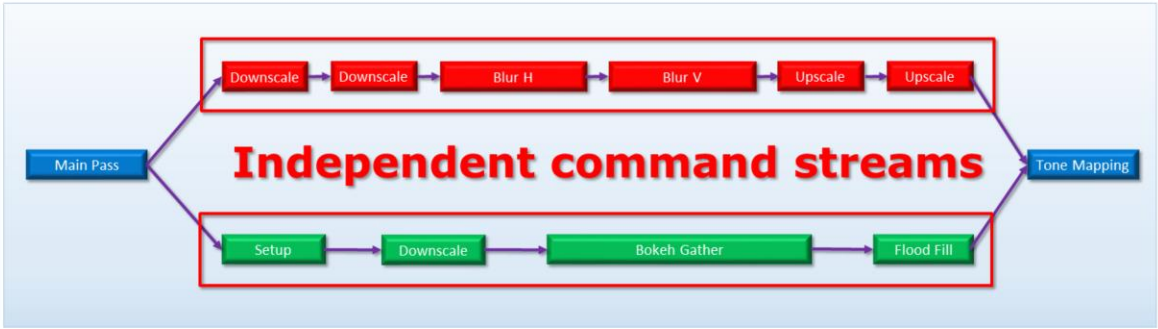
Thinking in CPU Terms

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

If we once again try to relate things back to the CPU world, it's not too much of a stretch to think of multiple front-ends as being roughly analogous to Simultaneous Multithreading (SMT), commonly referred to as Hyperthreading on Intel CPUs. This is a hardware feature of certain CPUs that lets two different instruction streams (threads) share the same set of execution resources of a CPU core. Typically the CPU will try to interleave instructions from

both streams in such a way that one thread can actually do some ALU work while the other is sitting around waiting for a cache miss. So really this has the same goal that we had in our examples with the MJP-4000: it’s trying to increase utilization by mixing in an independent command stream.

Real-World Example: Bloom + DOF

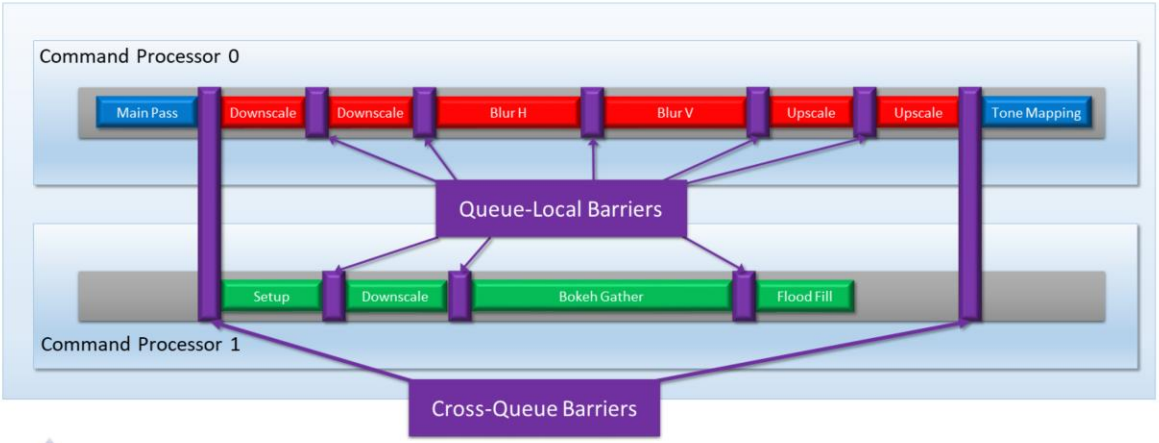


Now that we've gone over hypothetical examples, let's look at something you might actually see in a real game engine. This graph shows a fairly typical post-processing setup for a game. We start out with a main pass, which might forward rendered or use a deferred renderer to produce an HDR color buffer. This then gets fed into two different post-processing operations. On top, we have a standard bloom pass that works by downscaling, applying a

separable blur, and then upscaling back to full res. Meanwhile we also have depth of field pass going on below that. This might work by first calculating circle of confusion values from a depth buffer, then downscaling to half res, then performing a fancy bokeh-shaped gather, and finally performing a flood fill to fill in gaps in the sampling pattern. Both the bloom and the depth of field both feed into the tone mapping pass, which combines those two effects with the results of the main pass and applies tone mapping to bring everything down to LDR ranges.

These bloom and depth of field passes are an ideal case of having two independent command streams: they have dependencies within each stream, but no cross-dependencies between the two streams. This makes it a good fit for submitting them as parallel command streams to two front-ends.

Real-World Example: Bloom + DOF



Mapping this post-processing setup to two different front-ends gives us something like this. We now have the bulk of the work scheduled on the top command processor, with a parallel submission on the second command processor for the depth of field. The small purple bars in here are “queue-local” barriers, meaning they only synchronize on the work submitted by that particular command processor. But we also need some heavier cross-queue

barriers in order to make this work correctly, since we need to make sure that the depth of field submission doesn’t start processing until the main pass is finished.

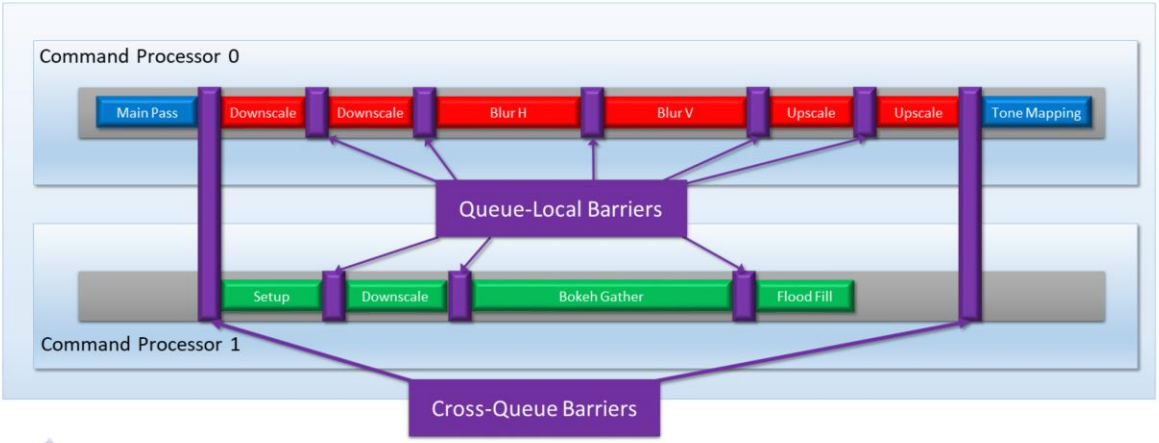
Submitting Commands in D3D12

- 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

If we're working with D3D12, then we have a rather explicit processor for building up command lists and submitting them to the GPU. Basically you'll record your command list(s), call ExecuteCommandLists to have them run on a particular command queue, and use fences to synchronize your submissions (potentially across multiple queues). When you submit to a command queue, what happens under the hood is that Windows has a GPU scheduler

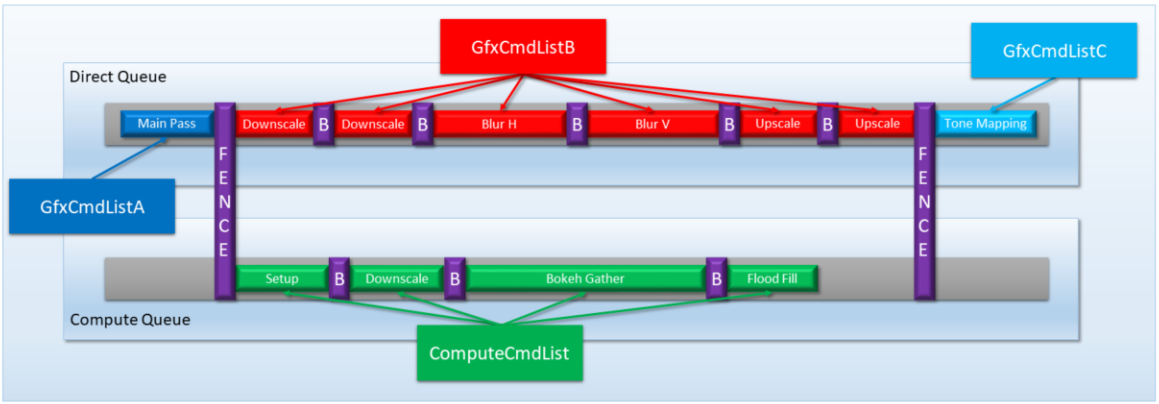
that will manage all in-flight submissions and schedule them on what the Windows Display Driver Model (WDDM) calls an “engine”. In this context an engine is basically a queue that the GPU driver exposes to the scheduler, which typically corresponds to an front-end on the hardware. Each engine has to specify which subset of functionality it supports, which maps exactly the direct/compute/copy command queue types available in D3D12.

Bloom + DOF in D3D12



Now if we take our bloom + depth of field example from earlier and map it to D3D12 submissions…

Bloom + DOF in D3D12



…we get something like this. We now have 3 command lists that will run on the direct queue, and possibly use full graphics functionality. Meanwhile we'll put the depth of field on its own compute command list, and submit on the compute queue towards the bottom. This maps fairly well to existing hardware, where its common to have 1 graphics front-end and multiple compute front-ends. Our "queue-local barriers" now turn into normal transition or UAV barriers,

and our cross-queue barriers turn into fences.

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

It's important to keep in mind though that submitting to multiple command queues doesn't guarantee that your submissions will actually execute in parallel like our MJP-4000 example! It really depends on a whole bunch of factors that include the OS scheduler, the hardware resources of the GPU, how the driver exposes things, and what type of command list you're submitting. It's actually very similar to threads on a CPU: executing things on multiple threads doesn't

guarantee that they’ll actually execute at the same time on multiple cores, you can easily just have two threads sharing the same core that switches back and forth.

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!

This ambiguity regarding command list execution is possible because D3D12 and the OS actually virtualize the command queues. Each ID3D12CommandQueue that you create doesn't correspond directly to some hardware queue or front-end: in fact you're allowed to created as many command queues as you want. The scheduler is responsible for taking all of your submissions and making sure that they execute on *some* hardware queue, of which

there might only be 1! The scheduler can do this by “flattening” your submissions if necessary, which basically amounts to serializing your multiple executions and running them sequentially. This sort of thing is possible because the API is setup in such a way that cross-queue synchronization is only possible in-between command list submissions, and not in the middle of them. It’s also made possible by the fact that the dependencies between submissions are made visible to the OS scheduler through your fences, which are actually quite heavyweight since they tie into the scheduler.

If you want to see what’s actually going on, you should definitely use your tools! GPUView is specifically designed to view command buffer submission activity on both GPU and CPU timelines, and can be utilized by taking an ETW capture (the easiest way to do this is to use Bruce

Dawson’s UIforETW tool: https://github.com/google/UIforETW). PIX for Windows also has a cool timing capture feature (https://devblogs.microsoft.com/pix/timing-captures/) that can show you your submissions are executing, and can also show you how your fences are causing waits. The vendor specific tools from AMD and Nvidia also have their own timelines and trace views that can be really helpful for seeing how your command lists are stacking up.

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

For any of those using Vulkan (or considering using it), you should be aware that the way Vulkan handles queues is actually a bit different. In Vulkan you can't simply create as many queues as you want: you instead have to query the device to see how many queues of a particular "family" are supported by the device, and then you have to explicitly bind to one of those slots. This gives the driver an opportunity to roughly inform you of the resources available

on the actual hardware, which lets you make some decisions based off of that. The downside is that your app needs to adapt on the fly and handle flattening itself for cases where there aren't as many queues available as you might expect, whereas in D3D12 the OS will just figure things out for you. And of course there's still no guarantee that the Vulkan queue slots explicitly map 1:1 with a hardware front-end, so there's still some virtualization possible here.

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

The kind of submission setup that I described earlier with the bloom + depth of field is commonly referred to as “async compute”, which you’ve probably heard of. This generally refers to having one main “graphics” submission, and one or more compute-only submissions to other queues. It was popularized by AMD, since their hardware tends to have 1 graphics front-end and lots of compute-only front-ends. However it also works on Nvidia hardware, which

also has multiple compute-only queues (Intel so far only exposes a single graphics queue and no other queue types). The goal is exactly what we did earlier in our MJP-4000 example: to throw in some independent command streams that file in the idle cores whenever there’s bubbles. They key to using it effectively is to identify strings of commands that are non-dependent, and then separating them into their own command lists and submissions. Since it’s all about filling in idle cores, you’re generally going to get the most bang for your buck by targeting points in your frame that have low shader core utilization.

Recap

Alright, we’re almost at the end! But first I want to give a quick recap to make sure that I emphasize the important points that I want you to take away from this talk.

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!)

The first major takeaway here is that GPU barriers are all about ensuring data visibility in the case of dependencies. On current GPUs this is generally going to involve some kind of GPU-wide thread synchronization operation, and could potentially involve some kind of cache action and/or data transformation (particularly in the case of decompression).

Since barriers are all about dependencies, it’s critical that you’re always thinking about your dependencies within a frame. Using a tool like DOT to visualize your dependencies can help you figure out where your choke points are, and also how you might re-arrange things to get better utilization. You can also consider using “frame graph” techniques to automatically discover your dependencies and submit your commands with optimal barriers

https://www.ea.com/frostbite/news/framegraph-extensible-rendering-architecture-in-frostbite
https://www.gdcvault.com/play/1024656/Advanced-Graphics-Tech-Moving-to
https://ourmachinery.com/post/high-level-rendering-using-render-graphs/

GPUs Aren't *That* Different

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

The next major takeaway here is that GPUs aren't really *that* different from multi-core CPUs. They have their own quirks, but at least in terms of thread synchronization you have to deal with a lot of the same issues. It's not too far off to think of your command processor as a hardware task scheduler, with the shader cores being the work cores/threads that a task scheduler will throw work at.

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

When it comes to barriers, they're almost always going to cause some kind of bubble where there's no work happening on your shader cores. The thread monster is always hungry, and you should do your best to minimize the periods where no threads are running. You can achieve this by being very careful about your barrier usage, and also trying to avoid scenarios where you're syncing on a draw or dispatch wither small numbers of very long-running

threads. And of course you should always always always batch your barriers, since flushing or syncing once is always going to be faster than doing it multiple times in a row.

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

Finally, if you're going to try to use multiple queue submissions to improve your utilization, then you should actually make sure that you're getting the desired result! With D3D12 in particular there's no guarantee that your submissions will actually run in parallel, and the hardware may have some quirks to prevent things from overlapping properly. Your GPU tools are definitely going to be your friends if you're attempting to do this.

Also, keep in mind that using multiple queue submissions isn’t going to magically increase the number of shader cores on the GPU. If you’re already saturating all of the cores, then you’re not going to be able to meaningfully increase the throughput. In fact you might actually just make things worse in that case by increasing cache contention, or even making it more difficult to properly time different passes since they’ll be sharing the GPU.

If you are going to try to use multiple queues, you’ll want to look for sequences of commands that are independent from other sequences of commands within your frame. Just be careful not to go crazy with the number of submissions and fences! Both can have a lot of overhead for both the CPU and GPU, and you so you’ll generally want to stay away from lots of submissions that only do a small amount of work.

That's It!

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

Alright, that's the end of the talk! I wanted to give a quick shout-out to the people from AMD and Nvidia that helped me get this talk into the conference, and also provided me with a lot of helpful advice and feedback. I also wanted to thank everyone at Ready at Dawn, especially those that came to my dry-run and gave me feedback.

Contact Info

- matt@readyatdawn.com
- mpettineo@gmail.com
- @mynameismjp
- https://mynameismjp.wordpress.com/
- https://github.com/TheRealMJP/GDC2019_Public
  - Includes pptx and PDF with full speaker notes

If you have any questions, comments, or feedback, please don't hesitate to reach out! I put both my work and personal email up here, along with my twitter handle, blog, and GitHub page.

If you're interested in a long-form in-depth version of the material that I presented here, feel free to head over to my blog where I have a 6-part series on this subject: https://mynameismjp.wordpress.com

/2018/03/06/breaking-down-barriers-part-1-whats-a-barrier/

Thanks again for downloading and reading this!