| COMMAND BUFFERS

Command buffers are the heart of the low-level graphics APIs. Most of the CPU time spent in DirectX®12 (DX12) and Vulkan® will be spent recording draws into the command buffers. One of the biggest optimizations is that an application can now multi-thread command buffer recording.

With the previous generation of APIs, the amount of multi-threading that could be done was severely limited to what the driver could muster up.

- Low level APIs allow multi-threading of command buffer generation. Using this can get better CPU utilization than higher-level APIs.
- The driver does not spawn extra threads for low level APIs. Applications are responsible for multi-threading.
- Avoid having too many small command buffers.
- Command allocators are not thread safe.
  - It is recommended to have a command allocator for each thread recording commands per frame.
  - The number of command allocators should be greather than (or equal to) the number of threads recording commands multiplied by the number of frames in flight.
- Command allocators grow in line with the largest command buffer allocated from them.
  - Try to reuse the same allocators for the same rendering workload to help minimize memory usage.
- Minimize the number of command buffers submissions to the GPU.
  - Each submit has a CPU and GPU cost associated with it.
  - Try to batch command buffers together into a single submission to reduce overhead.
  - Ideally, submissions would only happen when syncing queues or at the end of a frame.
  - Avoid using bundles and secondary command buffers.
    - They're likely to hurt GPU performance (they only benefit CPU performance).
    - Just fill the command buffers every frame.
    - If they are used have at least 10 draws per bundle/secondary command buffer.

DirectX 12

- Minimize state inheritance in bundles.

Vulkan

- Use the `USAGE_ONE_TIME_SUBMIT` on command buffers that are only submitted once before resetting. This maximizes the optimizations that can be applied.
- Avoid using `USAGE_SIMULTANEOUS_USE` on command buffers.
- Avoid clearing attachments inside a secondary command buffer.

| PIPELINE STATE OBJECTS (PSO)

All the smaller state structures where combined into a single state known as a Pipeline State Object. This allows the driver to know everything up front so that it can compile the shaders into the correct assembly. This removes any stutters that could happen in the previous API generations when the driver had to recompile shaders at the time of the draw call due to a change in state. This also allows easier state optimizations since there is no longer a need to track multiple small states.

- PSOs can take a long time to compile.
  - The driver will not spawn extra threads to compile.
  - Compiling PSOs is a great task for multi-threading.
  - Avoid compiling PSOs on threads with small stacks or in time sensitive code.
- Avoid just in time compilation of PSOs.
  - Consider using an uber-shader first and then compile specializations later.
  - Compiling pipelines using the same shaders simultaneously may result in serialization by lock contention.
- Use a pipeline cache to re-use PSOs across multiple runs.
  - It is recommended to share a single PSO cache to maximize the hit rate.
- Try to minimize the number of PSOs used.
  - PSOs use VRAM thus lots of them will quickly eat up your available GPU memory budget.
- Shader code does not need to be kept around after the PSO is created.
  - The blobs can be deleted to free up space.
- Minimize pipeline state changes when rendering.
  - The more similar the pipeline is to the previous, the lower the cost of the change.
  - Changing the bound PSO may roll the hardware context.
  - Sort draw calls by pipeline.
  - Group pipelines that use/don't use the geometry shader stage or the tessellation stages together.
- The driver does not do redundant state tracking.

DirectX 12

- Use pipeline libraries and not cached blobs.
  - Pipeline libraries use a fast path in the driver.
  - There is no need for locks when using a pipeline library. The DirectX runtime will handle synchronization.

Vulkan

- Minimize the number of dynamic states enabled in the PSO.

| BARRIERS

Barriers are how dependencies between operations are conveyed to the API and driver. Barriers open up a whole new world of operations by allowing the application to decide if the GPU can overlap work. They are also an easy way to slow down the rendering by adding too many barriers or can cause corruptions from not having correct resource transitions. The validation layers can often help with identifying missing barriers.

- Minimize the number of barriers used per frame.
    - Barriers can drain the GPU of work.
    - Don't issue read to read barriers. Transition the resource into the correct state the first time.
- Batch groups of barriers into a single call to reduce overhead of barriers.
    - This creates less calls into the driver and allows the driver to remove redundant operations.
- Avoid `GENERAL` / `COMMON` layouts unless required.
-
    - Always use the optimized state for your usage.

| MEMORY

Explicitly managing GPU memory has been exposed in both Vulkan and DirectX12. While this does allow many new optimization opportunities, it can also be hard to write an efficient GPU memory manager. That's why we created small open source libraries for that purpose for both **Vulkan** and **DirectX12**.

- Create large memory heaps and sub-allocate resources from the heap.
    - Recommended allocation size of heaps is 256MB.
    - Smaller sizes should be used for cards with less than 1 GB of VRAM
- Try to keep your allocations static. Allocating and freeing memory is expensive.
    - Use sub-allocations for dynamic resources.
- Help reduce memory footprint by aliasing transient resources.
    - Make sure to issue the correct barriers and only use necessary flags during resource initialization.
    - Improper use of placed/sparse resources can cause a hang and/or visual corruption.
    - Make sure to fully initialize (`Clear` , `Discard` , or `Copy`) placed resources to avoid hangs and/or corruptions.
- Use `QueryVideoMemoryInfo` and `VK_EXT_memory_budget` to query for the current memory budget.
    - It is recommended to only use 80% of the total VRAM to reduce the probability of eviction.
    - Don't call memory query functions per frame.
    - Instead use the `RegisterVideoMemoryBudgetChangeNotificationEvent` function to have the OS signal the app when the budget changes.
- Select the correct memory type for the resources intended usage.

    - Reading from and writing to system memory is slow from the GPU.
    - Use the Vulkan Memory Allocator (**VMA**) or the D3D12 Memory Allocator (**D3D12MA**) library to help simplify managing memory.
- Tiled/Sparse resources:
    - We don't recommend them based on their performance hit both on GPU and CPU.
    - Instead use sub allocations and the Copy/Transfer queue to defrag memory.
    - Never place high traffic resources in Tiled/Sparse (e.g. render targets, depth targets, UAVs/storage buffers)

DirectX 12

- Call MakeResident and Evict per heap when managing residency. Do not call these per resource.

Vulkan

- There is a small amount of guaranteed allocations. Always query for the maximum number of allocations supported first.
- On AMD cards, there is some GPU memory accessible from the CPU. This memory is known as device local and host visible.
    - This memory is great for updating constant buffers.
    - Don't overuse this memory. The driver also uses it.

| RESOURCES

A lot of hardware optimizations depend on a resource being used in a certain way. The driver uses the provided data at resource creation time to determine what optimizations can be enabled. Thus, it is crucial that resource creation is handled with care to profit from as many optimizations as possible.

- Set only the required resource usage flags.
    - Setting too many flags can disable hardware optimizations.
- Avoid using `STORAGE` / `UAV` usage with render targets.
    - This prevents compressing color on older generations of AMD GPUs, prior to GCN generation 3.
    - Alias the same memory allocation with a separate resource if you are only interested in memory savings.
    - This is not for data inheritance.
- Use non-linear layouts when available.
- Use buffer to image and image to buffer copies instead of image to image copies when uploading images.
- Avoid shared access on render and depth targets.
    - Make sure to correctly synchronize to help prevent corruption when moving between queues with exclusive access.
- Avoid using `TYPELESS` or `MUTABLE` formats on render targets, depth targets, and UAVs/storage buffers.
    - Explicitly use the type that will be used when reading and writing (`FLOAT` , `INT` , `NORM` , etc).
- Mipmapped arrays of render targets will not get compression optimizations.

- 24-bit depth formats have the same cost as 32-bit formats on GCN and RDNA.
    - Use a 32-bit format on AMD to get higher precision for the same memory cost as 24-bit.
- Use reversed depth (near plane at 1.0) to improve the distribution of floating-point values.
- Try to use the full range of the depth buffer.
    - Having all the values clumped up in a small part of the range can reduce depth test performance.
    - Improve depth distribution by having the near Z value be as high as possible.
    - Use depth partitioning for close geometry (e.g. player hands) to help improve this distribution further.
- Split your vertex data into multiple streams.
    - Allocating position data in its own stream can improve depth only passes.
    - If there is an attribute that is not being used in all passes, consider moving it to a new stream.
- Avoid setting vertex streams per draw call.
    - Updating the vertex streams can cost CPU time.
    - Use vertex and instance draw offsets to use data at an offset in the stream.
    - Vertex data can also be stored in SBOs/Structured buffers and fetched in a vertex shader instead of using vertex streams.
- Avoid using primitive restart index when possible.
    - Restart index can reduce the primitive rate on older generations.

| DESCRIPTORS

Descriptors are used by shaders to address resources. It is up to the application to provide a description of where the resources will be laid out during PSO creation. This allows the application to optimize the layout of descriptors based on the knowledge of what resources will be accessed the most.

- Try to minimize the size of the root signatures/descriptor set layouts to avoid spilling user data to memory.
    - Try to stay below 13 `DWORD` s.
    - Place parameters that will change often or need low latency in the front. This will help minimize the possibility of those parameters spilling.
    - Minimize use of root descriptors and dynamic buffers.
        - There is less validation on them. A GPU hang can happen when reading outside of the bounds of these descriptors. It is up to the application to check that all reads and writes are in the range of the resource when bound this way.
    - Avoid setting the shader stage flags to `ALL` on descriptors. Use the minimal required flags.
    - Minimize the number of descriptor set updates.
    - Reference resources that belong together in the same set.
    - Consider using a bindless system to help reduce CPU overhead from binding descriptors.
    - Order the draws by root signature when using multiple root signatures/descriptor set layouts.
    - The driver can embed static and immutable sampler into the shader, reducing memory loads.
    - Try to reuse samplers in the shader. This allows the shader compiler to reuse sampler descriptors when compared to binding multiple samplers that have the same value.

    - Only constants changing every draw should be in the root signature or push constants.
    - Avoid copying descriptors.
        - Unless it comes from a heap without the `DESCRIPTOR_HEAP_FLAG_SHADER_VISIBLE`
    - Prefer using `CopyDescriptorsSimple` / `vkUpdateDescriptorSet` over `CopyDescriptors` / `vkUpdateDescriptorSetWithTemplate` .
        - This function has a better CPU cache hit rate.

DirectX 12

- Descriptor tables cost 1 DWORD each.
- Root constants cost 1 DWORD each.
- Root descriptors cost 2 DWORDs each.

Vulkan

- Descriptor sets cost 1 DWORD each.
- Dynamic buffers cost 2 DWORDs each when robust buffer access is OFF.
- Dynamic buffers cost 4 DWORDs each when robust buffer access is ON.
- Push constants cost 1 DWORD per 4 bytes in the push constant range.

SYNCHRONIZATION

One large new addition to the graphics programmers tool box is the ability to control the synchronization between GPU and CPU. No longer is synchronization hidden behind the API. Submission of command buffers should be kept to a minimum as submitting requires a call into kernel mode as well as some implicit barriers on the GPU.

- Work submitted does not need semaphores/fences for synchronization on the same queue – use barriers in the command buffers.
    - Command buffers will be executed in order of submission.
    - Cross queue synchronization is done with fences or semaphores at submission boundaries.
- Minimize the amount of queue synchronization that is used.
    - Semaphores and fences have overhead.
    - Each fence has a CPU and GPU cost with it.
- Don't idle the CPU during submission or present: the CPU can continue preparing command lists for the next frame.

PRESENTING

When overlapping frame rendering with async compute, present from a different queue to reduce the chances of stalling.

DirectX 12

- Create a second direct queue dedicated to presenting.

Vulkan

- The compute queue can present on AMD.
- Check for hardware support prior to using a presentation mode.
  - V-Sync On: `FIFO_RELAXED` , if not supported, then `FIFO` .
  - V-Sync Off: `IMMEDIATE` .

CLEARS

AMD hardware has the ability to do a fast clear. It cannot be understated how much faster these clears are when compared to filling the full target. Fast clears have a few requirements to get the most out of them.

- Fast clears are designed to be ~100x faster than normal clears.
  - Fast clears require full image clears.
  - Render target fast clears need one of the following colors.
    - RGBA(0,0,0,0)
    - RGBA(0,0,0,1)
    - RGBA(1,1,1,0)
    - RGBA(1,1,1,1)
  - Depth target fast clears need `1.f` or `0.f` depth values (with stencil set to 0).
  - Depth target arrays must have all slices cleared at once to do a fast clear.
  - Use `Discard` / `LOAD_OP_DONT_CARE` to break dependencies when skipping a clear.

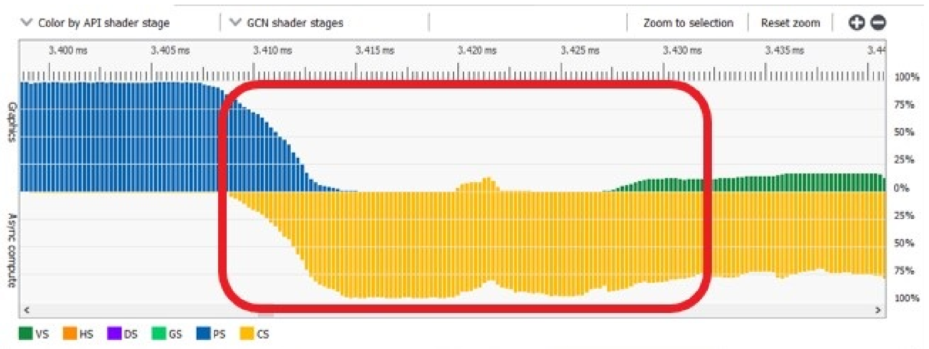
Vulkan

- Prefer using LOAD_OP_CLEAR and vkCmdClearAttachments over vkCmdClearColorImage and vkCmdClearDepthStencilImage.

ASYNC COMPUTE

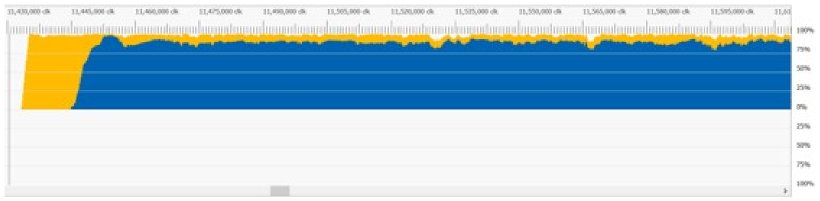
GCN and RDNA hardware have support for submitting compute shaders though an additional queue that will not be blocked from executing by the fixed function hardware. This allows filling the GPU with work while the graphics queue is bottlenecked on the frontend of the graphics pipeline.

- Async compute queues can be used to issue compute commands to the GPU parallel to the graphics queue.
  - This allows use of the shader resources when graphics can't produce enough work to fill the GPU.



Async compute fills compute units as graphics waves drain.

- Use async compute work to overlap frontend heavy graphics work.
  - Common overlapping opportunities include pre-Z, shadow rendering, and post-process.
  - The frame post-processing can be overlapped with the beginning of the next frames rendering.
- Async compute performs poorly when executed in parallel with export bound shaders.
- Smaller workgroups (64 threads) usually perform better than larger workgroups when run async.
  - This decreases resource requirements and increase opportunities for launching the shader.
- Any graphics work submitted after a compute dispatch or vice-versa can overlap if there are no barriers.
  - Small dispatches work better as pipelined compute vs async compute.
  - A large dispatch followed by a large draw is another place where pipelined compute works well.



Pipelined compute overlaps with a pixel shader

| ASYNC COPY

In addition to having a compute queue, GCN and RDNA also have dedicated copy queues. These copy queues map to special DMA engines on the GPU that were designed for maximizing transfers across the PCIe® bus. Vulkan and DX12 give direct access to this hardware though the copy/transfer queues.

- Use the copy queue to move memory over PCIe.
  - This is the most efficient way to move memory over PCIe.
  - The copy queue can also move memory around on the GPU and may be used to defrag memory asynchronously.
- Use compute or graphics queues to copy when both:
  1. The destination and source are on the GPU.
  2. The result is needed quickly.

SHADERS

General

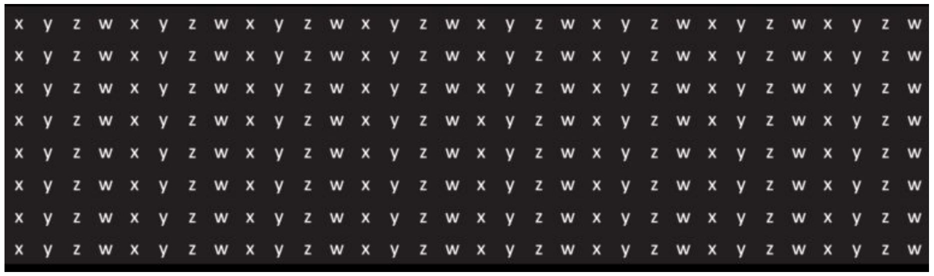
- Avoid arc functions (`atan` , `acos` , etc). They can generate code taking 100+ cycles.

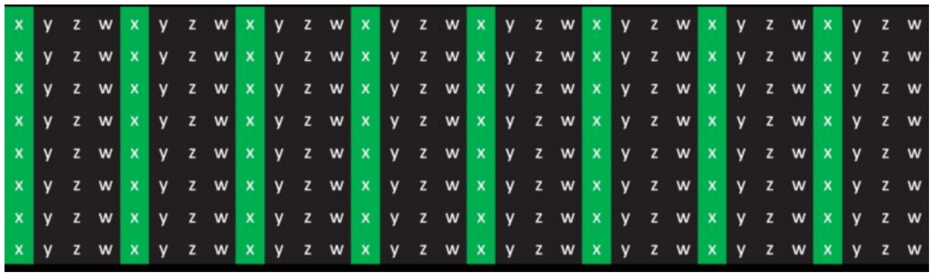
- Minimize use of transcendental functions (`sin` , `cos` , `sqrt` , `log` , `rcp`). These functions run at quarter rate.
  - RDNA can co-execute transcendentals at quarter rate.
  - Be careful with using tan as it will expand to 3 transcendentals: `sin` / `cos` .
  - Use approximations to help improve performance. **FidelityFX** has a library of fast approximations.
- GCN and RDNA support cross-wave ops via AGS, shader model 6, or SPIR-V subgroup operations.
  - Cross-wave operators are great for reduction problems such as prefix sums, downsampling, filtering, etc.
- Shader intrinsics can be used to reduce VGPR (Vector General Purpose Register) pressure.
  - FP16 can be used to reduce VGPR allocation count.
  - VGPRs can be scalarized though the readFirstLane intrinsic.
- When sampling a single channel, use `Gather4` to improve texture fetch speed.
  - While not reducing bandwidth, `Gather4` will reduce the amount of traffic to the texture units and improve cache hit rate.
  - `Gather4` may reduce the number of VGPRs required.
    - 8 VGPRs -> 2 VGPRs needed for addressing.

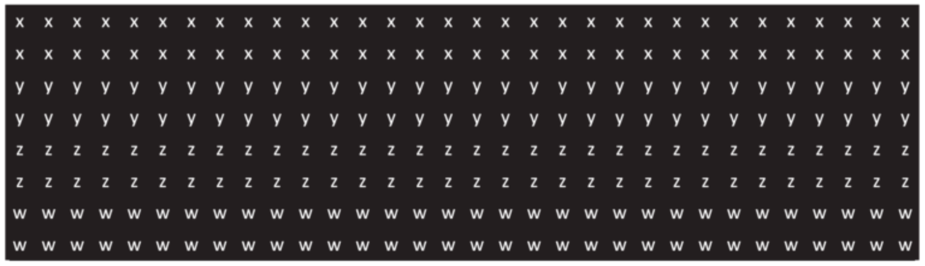
Compute shaders

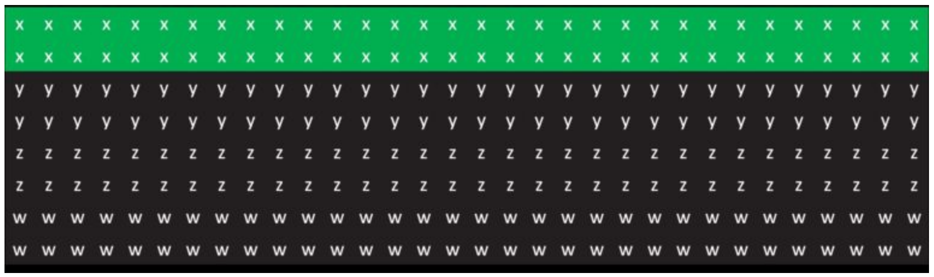
- GCN runs shader threads in groups of 64 known as wave64.

- RDNA runs shader threads in groups of 32 known as wave32.
- Unused threads in a wave get masked out when running the shader.
- Make the workgroup size a multiple of 64 to obtain best performance across all GPU generations.
- To help maximize bandwidth in compute shaders, write to images in coalesced 256-byte blocks per wave.
    - Rule of thumb is to have an 8×8 thread group write 8×8 blocks of pixels.
    - Using a swizzled layout of threads can improve bandwidth and cache hit rate.
        - **FidelityFX** has a helper function, ARmpRed8x8(), for this.
    - Thread group shared memory maps to LDS (Local Data Share)
        - LDS memory is banked on RDNA and GCN
        - It's spread across 32 banks
        - Each bank is 32bits (1 DWORD)
        - Bank conflicts increase latency of instructions.
        - Prefer a struct of arrays or add padding to reduce access strides and bank conflicts.


Float4 array

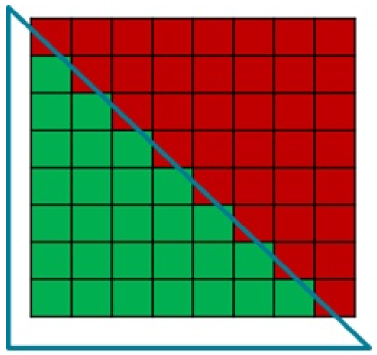

Reading X (8 bank conflicts)


Array of floats



Reading X (2 bank conflicts)

- Compute shaders can increase opportunities for optimizing using LDS or better thread utilization.
- Use a compute shader for full-screen passes.
    - This can provide up to 2% improvement over a full screen quad, depending on hardware configuration.
    - Removes need for helper lanes created at triangle edge.
    - Will usually improve cache hit rate vs a full screen quad.



A triangle will not fill the whole wave with a 8x8 tile

- Compute shaders tend to be faster in highly varying workloads compared to pixel shaders.
    - Pixel shaders can be blocked from exporting by other waves.

Vertex shaders

- Culling primitives using a geometry shader or cull distances is expensive.
- Cull primitives from the vertex shader by setting any vertex position to NaN.

Pixel shaders

- Using discard or modifying the output depth in any way inside pixel shaders prevents early-Z tests.
    - Avoid discard in long running shaders when there are other paths.
    - Use depth test equal to skip pixels that were discarded in a Z pre-pass.
- Minimize the number of render target outputs used.
    - Packing 2 render targets into a fatter format can improve performance over 2 smaller formats.
    - For example (2x RGBA8 -> RGBA16)
    - Avoid using RGB32 formats.

DEBUGGING

While debugging is not optimization it is still worth mentioning a few tips for debugging that will come in handy when optimizing rendering.

- The debug runtime/validation layer is your friend.
    - Titles must be warning free or risk breaking on some implementations or future hardware and drivers.
    - It might not catch all undefined behavior.
- Buffer markers help track down TDRs.
    - Write the markers into a buffer created with external memory. Otherwise, the buffer may be lost with the device.
    - Don't ship with a buffer marker per draw call. It can limit the amount of work in flight.
    - Use a marker per pass for shipping builds.
- Debug markers will show up in tools like RenderDoc, PIX and RGP.
- Implement an easy way to wait for the GPU to finish. It will help track down synchronization issues.

DirectX 12

- RGP can use PIX3 markers by using the PIX3.h file included with RGP.