GDC

D3D12 & Vulkan Done Right

Gareth Thomas
Developer Technology Engineer, AMD

Agenda

- Barriers
- Copy Queue
- Resources
- Pipeline Shaders

What is *not* in this talk…

- Async compute
  - Check out Async Compute: Deep Dive @ 13:20
- New features
  - Wave level programming: Stay in your seat! ☺

Barriers

- **Still** the #1 cause of poor performance over higher level APIs
- But barriers are hard to get right!

Barrier Issues

- Missing barriers
  - Corruption (Maybe)
- Too many barriers
  - Not batched
  - Not transitioning to the right state first time
- Incorrect barriers
  - Debug layers and GPU validation layers are your friends!
  - Catching 99% of issues
  - …and they are improving

Barrier Solutions

- Manual placement of barriers
  - Works well for simple engines
  - But gets complicated quickly
- Auto generation of barriers “behind the scenes”
  - Per resource tracking
  - Difficult to get right
  - Transition “on demand” can lead to lack of batching and often barriers in sub optimal places
- Simulate render passes on D3D12
  - Better portability

Barrier Solutions

- Frame graph
    - Analyse each pass to work out dependencies
    - Can then determine scope of each resource for memory aliasing
    - Case studies:
        - Tiago’s talk today
        - Yuriy’s talk on Thursday

*If you aren’t **looking ahead**, you probably aren’t making the most of D3D12/Vulkan*

Copy Queue

- Dedicated hardware designed specifically for copying over PCIE
    - Operates independently to the other queues

The rule is simple:

If copying from system memory to local, use the copy queue!

Copy Queue

- Ideal for streaming
- mGPU p2p transfers
- Make sure there is enough work on the GPU to ensure you don’t wait on the copy queue
  - Start the copy early as possible, ideally several frames, before it is required in local memory

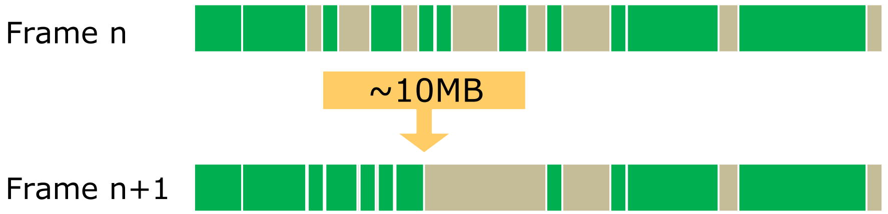
Copy Queue

- Don’t use the copy queue:
  - For local to local copies*
    - Use the graphics or compute queues
    - Copy queue runs at PCIE speed

*(*However, you can use the copy queue for “background” local to local operations like memory defragging)*

Memory Defragging

- Use the copy queue to move say 1% bw/frame
  - Leaves the graphics queue to continue rendering
  - Do this on frames where copy queue is not busy streaming



Pipeline Shader Management

- Try to minimize combinatorial explosions
  - Prune unused permutations early
  - Consider Ubershaders where appropriate
  - Root constants in D3D12
  - Specialization constants in Vulkan
- If building PSOs on the fly, build them well enough in advance

Resource Management

- You are in full control of resource management
  - You know how much memory is physically on the GPU
  - You know how much memory your game requires
  - Up to you to ensure local memory is not oversubscribed

Take action if you do end up oversubscribing

- Oversubscription can cause sharp fluctuations in performance
- Causes:
  - Other memory intensive apps gaining focus, browsers etc..
  - User changing resolution/quality settings
- Consider capping settings on 1GB, 2GB etc. hardware

How much memory is available?

- IDXGIAdapter3::QueryVideoMemoryInfo()
- Can lose budget dynamically

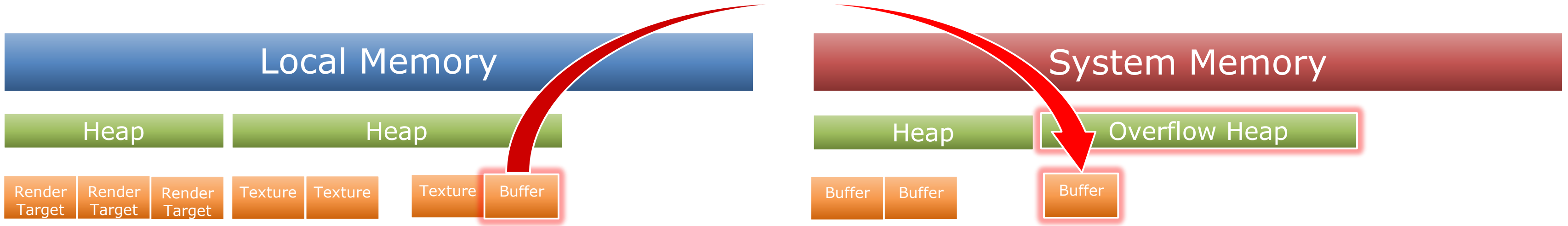
```
// get local memory info
DXGI_QUERY_VIDEO_MEMORY_INFO info = {};
m_adapter->QueryVideoMemoryInfo( 0, DXGI_MEMORY_SEGMENT_GROUP_LOCAL, &info );

// check against current local memory footprint
if ( m_totalLocalMemoryUsed > info.Budget )
{
    // take action!
}
```

Poll each frame or register for callback

What can you do to limit local memory?

- Move non performance-critical assets out of local memory
  - Into overflow heaps in system memory
- Drop top mip levels



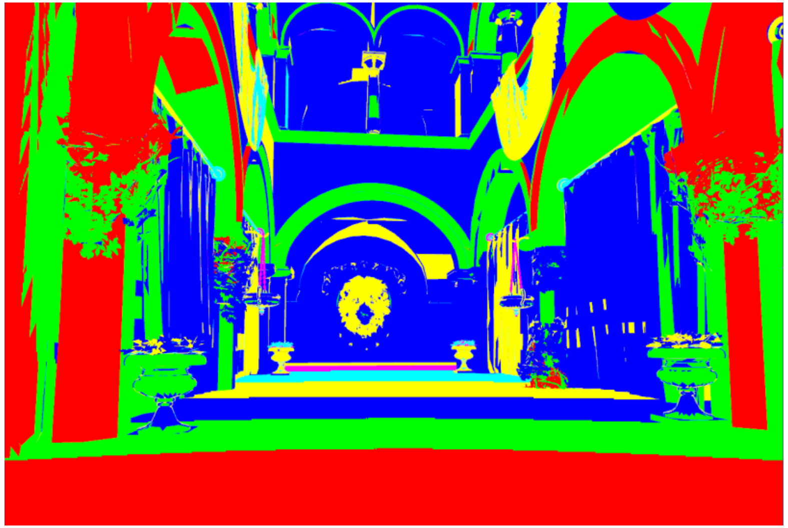
Moving assets out of local memory

- Free up local copy
- Understand the access pattern of your resources before moving to system memory
  - Read once
  - Predictable access pattern with high locality: good

Dropping top mips

Saves ~70% memory

- Little visual difference if done dogmatically
- No visual difference if done intelligently
- Easier to implement when textures are placed resources in a heap



Try testing with two instances of your title

MakeResident

- MakeResident can fail!
  - Must be handled
- MakeResident is a **synchronous** call
  - Does not return until every resource is ready
  - Batch it up and run it asynchronously
  - Small batches are inefficient -> lots of small paging operations
- Evict is less costly
  - Cost likely to be deferred to next MakeResident call

Conclusion

- Embrace the new concepts as first class citizens
  - Multithreading
  - Multiple queues
  - Render passes + frame graphs
  - Explicit resource management
- If you aren’t looking ahead, you probably aren’t making the most of D3D12 and Vulkan
  - Use your high level view to orchestrate your queues and barriers

Questions