Notes: Low-Level Optimizations in The Last of Us Part II by Parikshit Saraswat - SIGGRAPH 2020

https://www.youtube.com/watch?v=CvS6rX-67wA

Version 1.1

YouTube video Notes - Phani Srikar

Introduction

In the Video Parikshit talks about different optimizations done in the game over a number of passes:

- Subsurface Scattering
- Volumetric probe lighting
- Skiining
- GBuffer
- Water

Apparently this saved them around 2.1ms which is a lot of time to free up on the GPU in terms of rendering to be used by other teams such as gameplay/VFX/Animations etc. and stay within the frame budget of 33ms for PS4.

In these notes I'll try to dive deep into every sentence Parikshit says and delve into the topics he mentions like what's L2 hit ratio, export stalls, how to improve GPU stalling etc. and provide references wherever I can to avoid mistakes.

In future versions I'll try to add code snippets and GPU captures to validate my understandings.

Please correct if there are any theoretical misunderstandings. *Reach-out to me via email: phani.s2909@gmail.com*

Let's start...shall we?

Optimizing Subsurface Scattering

What is Subsurface Scattering?

Subsurface scattering a 3D graphics technique used to add more realism to materials [1]. It's a light transport property of light, where we model how the light gets scattered inside the model/surface of a material and how it behaves with translucent object to give a more natural and realistic feel to the material, typically materials like Hair, Skin etc. use subsurface

- *Skin diffuse shader had decent L2 hit ratio but was still memory bound*

what does it exactly mean?

So what's happening here though they are storing good amount of data in L2 and are accessing it with a good hit ratio, the majority of the bottle neck comes from accessing memory from a global scope, the

memory request spill over L2 and go to main memory causing bottlenecks in this case. If we need large data reads that exceed the L2 capacity, it will frequently access global main memory leading to latency, the global memory access can have worse cache misses and cause a lot of latencies, shader core stalling etc.

So try to keep memory spill to minimum in a shader. Don't access memory from different caches and spill it all over causing latency, coherency and shader core stalling bottlenecks.

- *Spill GPRs to LDS to reduce register pressure*

This is a really fun way of thinking, so we have other memory like LDS and GDS in the GPU in addition to the caches, these memory is available within the WGP - Work Group Processor and for inter-thread communication, and they can be used when the SGPRs and VGPRs are completely occupied and the register pressure is very high. So spill them to occupy the memory instead of registers.

ex. Instead of having many local variables, use `groupshared` *to spill the GPRs to LDS.*

How does this increase the occupancy?
Each thread requires a minimum number of registers to run, if the register pressure is less, we can have more thread dispatches hence increasing the occupancy of the shader. If the register pool is exhausted due to high register pressure, the GPU cannot run as many threads and thus decreasing the occupancy.

- *Even with reduced GPR pressure it was a low-occupancy shader, what they mean is that it won't get schedule with other items and run better, if it can be run then the memory-latency can be hidden when scheduled frequently for execution*
- *Switching from Compute to Pixel Shader helps with the wavefront distribution and reduces memory stalls*

Occupancy is directly proportional to GPRs, then why is it still low occupancy? That's because of the GPU scheduler not giving enough priority due to its memory access latencies, one way to hide this is to frequently run it along with other workload, what this means is that, parallelize the work with a shader that has a lot of ALU operation so that the delay for memory access is hidden by this shader.

Another trick is to use a pixel shader instead of a low-occupancy compute shader, the wave distribution of the workload can be improved. How so?

1. Pixel shaders runs in screen space per every pixel which naturally forces them to launch maximum wavefronts and force them to run at maximum occupancy possible
2. **Memory latency is hidden by switching between wavefronts when one wavefront is stalled waiting for memory**
3. All the write of a Pixel Shader goes through the DCC color cache instead of GPU L2$, which frees up the L2$ for more read operations

- *The Shader run by itself run slower than the CS due to ordered export stalls*

Note: Don't always do this, If your shaders has a lot of branching it's better off as a CS due to early out and we don't want the PS to stall until all the export writes are done, so thread carefully when choosing between a PS or CS version of the shader, in this case you don't need to do async optimization like mentioned above, keep it as-is, cause the PS version will be slower in this case

- *While the PS is slower by itself, it has much better wavefront distribution*

- *So we were able to hind the export stalls and and memory latency and fill those gaps with Async ALU and memory operations*

Thought export stalls were making the PS slower, due to higher wavefront distribution we could have more memory operation triggered and these 2 things were hidden under a async distribution of other shaders that had enough ALU and memory operations to finish

- *Pattern of pixels in a PS wavefront is not like 8x8 tile of a CS*

This is a major issue if you're dealing with lights and using wave intrinsics to get neighboring light lists within the quad of a wave. Pixel shaders process fragment quads (2x2 groups of adjacent pixels) to calculate derivatives for operations like mipmapping in a non-uniform fashion (that's how rasterization pattern works), whereas a wavefront is formed by mapping multiple quads together, they run sequentially in groups of 8x8, 16x16 etc. So if a thread is outside the triangle it becomes inactive causing **wave divergence**

Optimizing Probe Lighting

- *The application of light probs is a memory intensive shader while also happens to trash the caches quite a bit because of random access patterns*
- *Shifting to PS made more L2 available for reads since all write go through color cache now*

This is the same optimization as above which helped reduce L2 occupancy and speed things up. Again due to branching and export stalls this PS will be slower than its CS version. So the good'ol trick is to hide this latency with an async shader and this will eventually result in a faster version than the CS version

- *Since the volumetric probe lighting shader was trashing our caches, we benefitted from reducing it's occupancy from 4 to 3 and clustering our memory reads together as early as possible*
- *Consider decreasing it's occupancy by increasing the lifetime of VGPRs*
- *Hide latency of memory reads by using the ALU in each wavefront instead of trying to run more wavefronts adding to cache trashing*

Note: In the second point it's mentioned we need to decrease occupancy? hmm...weird right? NO! checkout this talk the from the graphics programming conference, that explains occupancy in detail and why it's not always good to have high occupancy[5] - `https://www.youtube.com/watch?v=sHFb5Xfwl9M` Think you can come up with your own reasons, I think this is why it's inefficient to gave too high of a occupancy:

- Too many threads, implies too much scheduling overhead for the Shader Processor
- Too much competition for L1$/L2$ cache between wavefronts, this leads to frequent evictions and reloads of cache lines, reducing memory access efficiency
- Can quickly extinguish the register pool and causing implicit spill to slower memory like LDS/GDS

So while you need the occupancy to be higher, keep a good balance of it so that it's no through the roof, that it actually ends up slowing the shader!. This tradeoff works best for arithmetic-bound workloads.

So we know that the more registers get used the less we have in register pool and the occupancy decreases...we talked about this. Now how to increase the lifetime of VGPRs? It's done by clustering all the memory reads together as early as possible, this way more memory reads will use more VGPRs and increase their memory lifetime, and as parikshit says they'll apparently end up living for more duration than they

were actually needed.

I don't get the third point thought...will re-visit this later! or will try to contact parikshit for more info. But here's a crude take that might be true or completely BS:
Latency hiding through more wavefronts becomes diminishing returns - what I mean is that while adding wavefronts hides latency by keeping more threads active, beyond a certain point, the benefits plateau due to bottlenecks like memory bandwidth and cache capacity, so in this case it's better not to launch any more wavefronts and stick to the current setup. So while the memory is being loaded do some ALU computation instead to the keep the thread active and then resume back once the memory is ready. **Keep it more ALU intensive to hide memory latency within the wavefront**

So how do we know whether to increase or decrease the occupancy? When to hide the memory stalls in async or ALU? the answer is simple **PROFILE! PROFILE! PROFILE!** and see what works best for your case in hand.

1. If your shader is less ALU-intensive and extensively memory-bound, you can **increase its occupancy** by scheduling it alongside other asynchronous passes, such as using it as a PS. This allows high occupancy to hide memory latency stalls and export stalls by overlapping its execution with other workloads.

2. If your shader involves a significant amount of ALU operations and is moderately memory-bound, running it with excessively high occupancy can lead to issues such as cache thrashing, increased contention for registers/L1$/L2$ caches (resource retention in caches with memory stalls is bad) etc. Instead, **decreasing occupancy** by limiting active wavefronts ensures better resource utilization and avoids register spilling, allowing the shader to execute with reduced latency and less cache trashes and hide these memory reads/writes while executing the ALU operations.

Optimizing Skinning

- *Instead of having the threads read the same bone again and again they tried loading all the bones into LDS, then have memory bariers for the thread group and then proceed execution to save time*

- *we had several different shaders and not enough waves*

So for them doing this sync waits on reading bones was leading to slow code because of many shader switches, they were better off with many memory reads. But if you don't have many such shaders and your shader has good amount of waves running, you can store the bones in a intermedia LDS buffer and run a large enough workgroup. The generic approach works very well fine here so saving memory reads via LDS and caching for most purposes.

If a lot of threads in the same WG need to access the same shared memory consider using larger thread group sizes, also is you're having massive workloads use larger thread group size and for smaller and irregular work loads, use smaller workgroup sizes.

- *We had a lot of L2$ cache trashes for recomputing normals after skinning*

- *It didn't have much ALU so clustering all memory reads didn't help much*

So what's happening here, is like we previous optimizations when we have memory latency we can hide it with ALU instructions or async work, but here the shader didn't have much ALU operations to hide the latency... and no async work to hide with either... so what to do now? Decrease the occupancy of the shader, why? cause this shader was disrupting L2$ soo much and was messing wither other async work

shaders (since it's a low ALU shader we hide it's latency under async work - jogging memory), such a weird edge case scenario!

IDK didn't get this part much...will update later...too niche for me to understand the intuition behind this approach

Optimizing GBuffer

- *GBuffer is huge so there's a lot of async work that goes on with it for soo long it just runs on it's own for some time*
- *We could never hit full GPR allocation in our GBuffer pass due to being parameter cache bound*
- *since PS can't fill the GPRs might as well spawn extra VS to do some ALU and mem ops*

Okay the second sentence is daunting! What does it mean? - it means that the shader is bound by caches like Texture cache, Constant cache, etc. [6] where it waits on reads from parameters like texture reads and uniforms buffers etc. so it's hard to optimize it further and during this time we can just invoke more VS invocations and have it overlap with PS invocation and basically do the VS mem and ALU ops for free, and maybe even push some ALU from PS to VS as well.

Glossary

Some definitions of the terms used in the doc:

Lane/Thread: Always and almost used interchangeably, a lane corresponds to a single thread of execution inside a SIMD execution group.[3]

Wave/Warp(NVidia)/Wavefront(AMD): A set of lanes executed simultaneously. Typically a group of 32x32 or 64x64 threads is considered a Wavefront.[3]

Helper Lane: Lanes executed in a Pixel Shaders for the purpose of gradients in PS quads. The output such lanes are discarded and they have restricted access to memory.[3]

Inactive Lane: Lane in the WG not being executed, possibly dues to control flow or insufficient memory or registers.[3]

Active Lane: Lane in the WorkGroup that is currently being executed.[3]

Occupancy: The ratio of number of active lanes to inactive lanes in a wavefront. For ex. we can manually visualize it using `WaveActiveBallot` wave intrinsic function in D3D12 in SM6.0 or above.

Quad: A set of adjacent lanes corresponding to a 2x2 square. Used to calculate gradients by diffing in x and y direction, and a warp may have many such quads.

L2 Hit ratio: in a GPU we have a L1$ followed by L2$, [2] L2$ hit ratio is the percentage of successful L2 memory requests made, i.e. without and cache misses.

Memory Bound: It means the perf at this point is bound by the memory operations (such a read/write etc. from main memory/cache) and not by the ALU instructions.

Ordered export stalls in a PS: All the writes of a PS are ordered, so if a PS doesn't write anything it exports 0's, it must wait until all the PS invocations are done and this will cause export stalling of the GPU.

Wave Divergence: Caused when different instructions are to be executed by different threads in a same wavefront due to conditional branching, causing some threads to idle, decreasing the overall efficiency.

References

[1] https://en.wikipedia.org/wiki/Subsurface_scattering

[2] https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html?highlight=l2#l2-cache

[3] https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12SM6WaveIntrinsics

[4] https://docs.unity3d.com/Manual/LightProbes.html

[5] https://gpuopen.com/learn/occupancy-explained/

[6] https://www.rastergrid.com/blog/gpu-tech/2021/01/understanding-gpu-caches/

[7] https://modal.com/gpu-glossary