Low-level Optimizations in *The Last of Us Part II*

Parikshit Saraswat
Naughty Dog, LLC

ABSTRACT

Developing a game that satisfies high expectations on visual quality while running at 30 fps on a 6 year old console platform, *PlayStation 4*, is quite a challenging task. The majority of the world of *The Last of Us Part II* contains alpha geometry which has poor rendering pipeline performance and full-screen expensive shaders with heavy resource usage which required us to use plenty of hacks and tricks to push the hardware to its limits and make sure we don't have to compromise needlessly on the visual fidelity of the game. This talk will mainly focus on case studies from the actual game highlighting the most expensive parts of our game's frame and how we were able to come up with solutions to these performance hurdles.

CCS CONCEPTS

• **Computing methodologies** → Rendering.

KEYWORDS

games, hardware, mathematics, production, real-time, rendering

ACM Reference Format:
Parikshit Saraswat. 2020. Low-level Optimizations in *The Last of Us Part II*. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks (SIGGRAPH '20 Talks), August 17, 2020.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3388767.3407359

SIGGRAPH '20 Talks, August 17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7971-7/20/08.
https://doi.org/10.1145/3388767.3407359

1 INTRODUCTION

The rendering engine of *The Last of Us Part II* is heavily optimized on a higher level with a render graph to schedule execution of various passes based on several dependencies and pass-specific hints about its resource usage and execution bottleneck. However, to make sure these passes also run as optimally as possible on the hardware, we had to make several changes to the rendering setup. The main goal of this talk is to show case studies of what specific challenges we were facing and which optimizations we used to tackle them.

2 OPTIMIZATIONS

2.1 Instruction and Wavefront Scheduling

Controlling how the compiler schedules instructions can be very useful for increasing total number of asynchronous wavefronts running or decreasing latency of rendering passes that run exclusively. Similarly, depending on whether the pass is thrashing the L2 cache or not, we can decide whether to increase or decrease its occupancy [Mah 2013] by utilizing the local data storage that is present on each compute unit of *PlayStation 4*'s GPU. This way, we can exclusively limit occupancy of one pass while making sure that we leave the register file as empty as possible for other asynchronous passes to run with it provided they don't require the shared memory themselves.

Our skin rendering shader was initially running for all the pixels with an early out for non-skin pixels, however because of heavy use of the register file and local data storage, none of the asynchronous passes were being scheduled with it leading it to run exclusively instead of sharing the unused resources during its execution. By running it as a pixel shader instead of a compute shader while moving over the bottleneck to the rendering pipeline's stencil testing units, we were able to get multiple asynchronous passes running with our skin shader and maximize instruction issue rate during its execution. This, however, came with its own challenges due to different screen-space layouts of pixel and compute wavefronts and the specific tiling restrictions of our skin shader, which required us to perform some precomputations on the CPU side in order to decide whether to perform skin rendering on a pixel shader or a compute shader.

2.2 Geometry Pipeline

The pixel shaders of *The Last of Us Part II* make heavy use of interpolants so while having efficient clipper setup helped us to minimize rendering pipeline cost of our ocean drawcalls making sure we never run out of pixel wavefronts to run on the compute units, the rest of our geometry rendering is still either bound by the per-wavefront storage which contains the interpolant data or by the depth testing in case of alpha geometry due to heavy overdraw. Increasing performance of these geometry passes and also the asynchronous passes running with them was quite challenging and required various content changes to maximize throughput of the rendering pipeline. The vertex shaders of our geometry buffer pass were bound by the limited per-vertex storage in the hardware as opposed to occupancy and the data allocation scheme used by them during export. Minimizing their resource usage and increasing the number of simultaneous vertex wavefronts increased the performance of our geometry passes while making more resources available to the other asynchronous passes running with it.

In our depth only pass, we run cheap pixel shaders for the alpha rendered geometry and expensive ones for our snow rendering. Shader cost and efficient depth testing setup is key for choosing whether to pick the compute units as the bottleneck or the depth testing units in the rendering pipeline. By carefully deciding when to use EarlyZ, LateZ or both, we were able to maximize wavefront throughput in alpha geometry with huge overdraw and minimize compute unit resource usage in case of large geometries which can undergo efficient primitive culling in the rendering pipeline resulting in significant improvements in our depth only pass which needs to complete as soon as possible in order to be able to dispatch the rest of the frame's passes.

Since *The Last of Us Part II* needed to support over a thousand materials, our geometry buffer pass ends up with a lot of short drawcalls with different shaders resulting in poor instruction cache hit ratio and gaps of 300 cycles throughout the shaders. Using CPU side geometry and shader analysis, it's possible to manually control which compute units execute which drawcalls to maximize instruction cache hit ratio and increase instruction throughput.

2.3 Compute Pipeline

There can be waits before issuing instructions for several reasons such as branching [Drobot 2014], instruction execution rates [Persson 2014] and instruction fetch stalls but it's possible to be bound by the texture unit inside each compute unit as well which introduces waits before memory read and write instructions. This could be either due to limited memory bandwidth or expensive filtering. In the latter case, it's possible to perform filtering on the compute unit manually to increase instruction issue rate. Our vertex deformation shaders are heavily ALU bound. Usually trading off ALU with memory using local data storage and threadgroups is a good idea. However, in our case, since there are too many different types of deformation shaders and the amount of work per unique shader is too little, using threadgroups forces the hardware to create synchronization waits. This results in decreased instruction issue rate and creates instruction fetch stalls in cases where the amount of wavefronts in the threadgroup is low. Choosing ALU over memory resulted in increased wavefront throughput in case of our vertex deformation shaders.

2.4 Memory Bandwidth

Some of the passes which render sparsely on the screen like user interface and then require the result to be read in another pass were optimized using custom bilinear filtering to make sure we don't waste any memory bandwidth clearing the input texture and reading in pixels which haven't been written to in the first place. Similarly, we run all our rendering passes which require reading and writing heavily to memory on pixel shaders as their output goes through separate color/depth buffer caches increasing hit ratio of the L2 cache during reads.

3 CONCLUSION

To make sure *The Last of Us Part II* achieves the maximum visual fidelity while running at our target frame rate of 30 fps, we used various optimizations to fully utilize the *PlayStation 4* hardware and being one of the bestselling consoles, this information along with case studies from the actual game would help the real-time rendering community a lot to push the hardware to its limit.

REFERENCES

Michal Drobot. 2014. Low-level Optimizations for GCN: Hacking the Next Generation. In *Digital Dragons 2014.*

Layla Mah. 2013. Powering the next generation of graphics: AMD GCN Architecture. In *GDC Europe 2013.*

Emil Persson. 2014. Low-level Shader Optimization for Next-Gen and DX11. In *GDC 2014.*