Posts

Shader Execution Reordering: Nvidia Tackles Divergence

May 16, 2023 clamchowder 3 Comments

In a previous article, we covered Cyberpunk 2077's "Overdrive" raytracing mode. Rasterization and raytracing are both parallelizable, but raytracing tends to be less predictable. That makes it difficult for GPUs to bring their full parallel processing power to bear. Divergence is one such culprit.

From a developers' point of view, GPU programming APIs look like you're spawning off a lot of independent threads. But under the hood, these threads are organized into groups called wavefronts or warps, that run in lockstep. This allows the hardware to use one program counter and instruction opcode to do math on many data elements, taking advantage of more parallelism without massively bloating instruction control hardware. For example, take the pixel shader below. To render at 4K, it dispatched 8,294,401 (3840*2160 + 1) threads, which are organized into 259,201 wavefronts.

Search …

Sort by
Relevance

Archives

December 2023
November 2023
October 2023
September 2023
August 2023
July 2023
June 2023
May 2023
April 2023
March 2023
February 2023
January 2023
December 2022
November 2022

Radeon GPU Profiler view of a frame from *Titanic and Honor Glory* demo

However, sometimes threads fail to run in lockstep even though they are supposed to, and that's when parallelism through wavefronts starts to fall apart. If some threads within a wavefront take a conditional branch while others don't, GPUs have to cope by executing both sides of the branch and disabling the lanes that the incoming instructions don't apply to. Thus, control flow divergence can drastically reduce execution unit utilization.

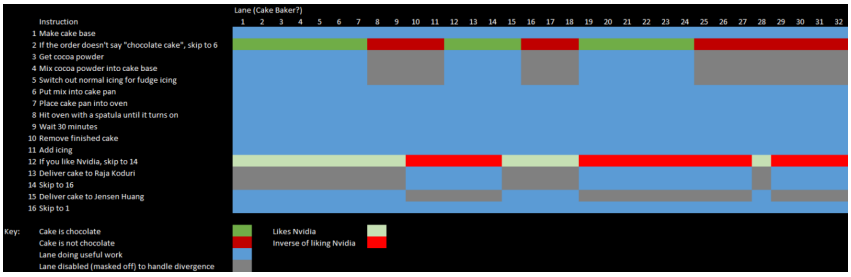


Illustration from our previous article demonstrating how control flow divergence can impact execution unit efficiency

Memory accesses can suffer from divergence, too. Even though instruction sets allow developers to address memory at the byte level and grab varying data sizes, the hardware handles memory in fixed-size, aligned chunks. "Aligned" here means the address is a multiple of something. For example, a 64-byte aligned address would be a multiple of 64, and the low 6 bits of the address are zero.

If a wavefront accesses a contiguous, aligned memory region, the cache can operate very efficiently. For example, Ada and RDNA's L1 caches can both deliver 128 bytes per cycle. A 32-wide wavefront can therefore get a 32-bit value loaded for each of its lanes with such an access. But if some

of the lanes split off and read from somewhere far away, a single memory access instruction in the wavefront could turn into several cache accesses under the hood, bringing down cache bandwidth efficiency.

Divergence in Cyberpunk 2077 Overdrive

To be clear, non-raytracing workloads can suffer from divergence too. Conditional branches and divergent memory access are not unique to raytracing. We can quantify the effect of divergence on performance by looking at different calls within a Cyberpunk 2077 Overdrive frame.

Action	Duration	Average Active Threads/Warp	Coherer
DispatchRays	26.37 ms	11.6 out of 32 possible	36.3%
DispatchRays	2.37 ms	13.1 out of 32 possible	40.8%
Draw 1943-2243	0.65 ms	16.8 out of 32 possible	52.4%
Draw 592-703	0.39 ms	28.3 out of 32 possible	88.4%
ExecuteCommandLists	2.27 ms	22.1 out of 32	69.1%

0 (4)	possible

From the profile of JigJig street with path tracing enabled (with SER enabled by default), collected by Cheese on NVIDIA RTX 4070

Divergence hits raytracing hard. Vertex shaders and compute also see notable execution efficiency losses from divergence. In contrast, pixel shaders barely suffer at all, which is noteworthy because pixel shading tends to dominate rasterization workloads, particularly at higher resolution.

Shader Execution Reordering

NVIDIA mitigates both of the divergence problems with Shader Execution Reordering, or SER. SER reorganizes threads into wavefronts that are less likely to suffer from divergence. Cyberpunk 2077's developers have implemented SER in the "Overdrive" mode, and is enabled by default. It can be disabled by setting a "EnableReferenceSER" flag to false via mods. Thanks to a user on Reddit, we were able to get traces with SER on and off.

Action	Duration	Average Active Threads/Warp	Coherence
DispatchRays, SER On	18.2 ms	14.2	44.2%
DispatchRays, SER Off	24.12 ms	9.7	30.4%

From traces collected by SebaRTX on Reddit, using NVIDIA RTX 4070 Ti. Traces were collected for a different scene than the one we previously profiled.

With SER, runtime for DispatchRays calls decreased by 24%, so SER is a large overall win for performance. NVIDIA's efforts should be applauded here, as SER is able to increase the number of active lanes in a wavefront by 46%. Besides losing less performance from control flow divergence, Ada was able to issue instructions to the execution units more

often when SER was on. I'm not sure why that's the case. Maybe SER introduces overhead that consumes instruction issue bandwidth.

Evaluating the data-side impact is harder because we don't get direct metrics from Nsight. In comparison the L1 load/store utilization is mostly the same, even though the SER version had shorter runtime. It was lower compared to SM issue utilization when SER is on. That means the L1 had to service fewer requests for approximately the same amount of work.

Action	SM Issue Active	L1 LSU Data-stage Throughput	L1 LSU Writeback-Stage Throughput	L1 Text Data-St Throug
DispatchRays, SER On	35.5%	24.8%	12.7%	2.2%
DispatchRays, SER Off	23.9%	24.6%	10.7%	2.0%

Writeback and texture throughput see small increases with SER, which is expected because jobs finish faster with SER on, which demands more throughput per unit time.

Even though SebaRTX profiled a different scene, cache hitrates are similar. Again, NVIDIA's L2 cache does an excellent job and catches the vast majority of memory accesses. VRAM bandwidth utilization is very low, which is a good thing.

GPU	Action	L1 Hitrate	L2 Hitrate	VRAM Throughput
RTX 4070 Ti	DispatchRays, Seba's Scene, SER On	63.8%	97.3%	10.8% read, 2.2% write

RTX 4070 Ti	DispatchRays, Seba's Scene, SER Off	64.2%	97.8%	7.2% read, 0.7% write
RTX 4070	DispatchRays, JigJig Scene, Assumed SER On	67.6%	97.1%	5.6% read, 2.3% write

Hardware Implementation Thoughts

AMD documents their raytracing optimizations with a lot of low-level detail. Their ISA manual describes intersection test instructions and (for RDNA 3) LDS instructions for traversal stack management. Then, the Radeon GPU Profiler shows these instructions in action within raytracing kernels. NVIDIA, on the other hand, only provides high-level descriptions. But we can try to read between the lines within their documentation, and SER seems to work like this:

1. Shader calls `NvReorderThread()` and provides a "key", or coherence hint that acts like a sort key. SER therefore needs explicit developer support, and doesn't work transparently like out-of-order execution on a CPU;
2. "Live" variables in the shader are written to the memory subsystem, and the threads are halted. That means SER incurs some overhead, and part of that overhead has to do with spilling and reloading variables;
3. Hardware sorts threads by their respective keys, thus putting threads with the same key close together;
4. Wavefronts are created out of the sorted list of threads and scheduled back onto the SM(s).

While NVIDIA doesn't explicitly say so in their SER whitepaper, their compiler is probably responsible for spilling the live register state to memory, and reading it back when thread execution resumes after SER. The GPU command processor would then have to move very little

state data between SMs. It would have to preserve the thread's program counter, its index within the group of threads dispatched in that call, and its index within a workgroup, but that's probably about it. With that information, the thread would be able to find its state from memory and reload it.

```
while(..)  // loop over light bounces
{
    HitObject hit = TraceRayHitObject( ray, payload );

    // Before we reorder, figure out if this will be the last loop iteration,
    // and encode that information in a coherence hint bit.
    float albedo = hit.LoadLocalRootTableConstant( 0 );
    bool done = russianRoulette( payload, albedo ) || bounceCount >= maxBounces;
    uint coherenceHints = done ? 1 : 0;

    // Reorder and shade
    ReorderThread( hit, coherenceHints, 1 );
    InvokeHitObject( hit, payload );

    // Because we included the 'done' flag in the coherence hints, chances
    // are good that all threads in the warp will make the same decision about
    // whether or not to break out of the loop.
    if( done )
        break;
}
```

Pseudocode example from Nvidia's Shader Execution Reordering whitepaper, showing ReorderThread called within a loop

NVIDIA is apparently able to do all this quite efficiently, because they show examples where SER is invoked many times within a shader. SER's mechanism for redistributing work has a lot in common with how an operating system moves threads between CPU cores. Migrating threads within a performance-critical loop would be terrible on a CPU though.

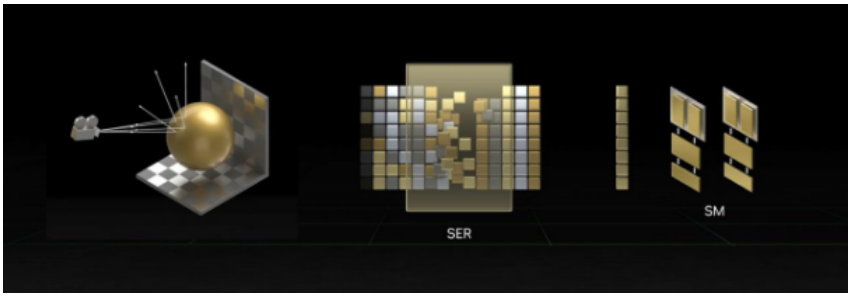
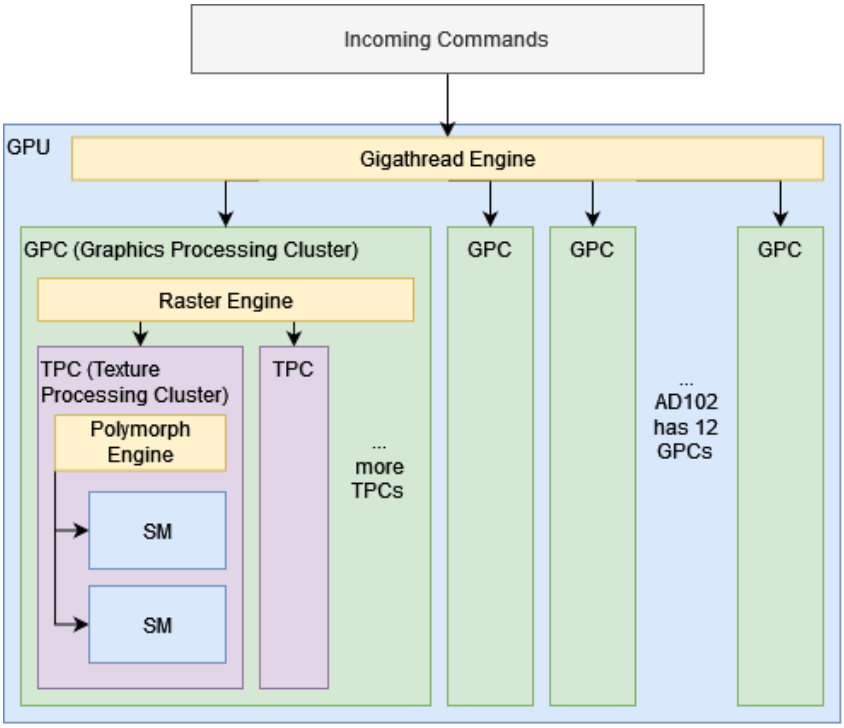


Illustration of SER from Nvidia's whitepaper

However, the whitepaper does not tell us how or why SER does its reordering so efficiently. It's time for reading between the lines and doing some more educated guessing.

Redistributing Work – How?

`NvReorderThread`'s coherence hint is specified in two parts: the key itself, and a number indicating how many bits to consider within the hint. That's quite an interesting detail, because sorting is typically an expensive operation. The best general purpose sorting algorithms like Quicksort and Mergesort generally run in O(n*log(n)) time, where n is the number of elements. Though with specifying the number of bits to sort with, makes radix sort particularly attractive. Radix sort runs in O(m*n) time, where n is the number of elements and m is the number of digits (bits). If only a few bits need to be considered, radix sort basically runs in linear time. Furthermore, Nvidia has already worked on creating fast GPU radix sort implementations.



Traditional work distribution points in an Nvidia GPU

Another question is where Nvidia is redistributing work. Normally, work gets distributed at three levels in Nvidia GPUs. The "Gigathread Engine" distributes work to the Graphics Processing Clusters (GPC). With rasterization workloads, each GPC gets a rectangle in 2D screen space. Then, each GPC has a raster engine that sends work to the (badly named) Texture Processing Clusters (TPCs). Finally, each TPC handles some work redistribution between shader stages when tesselation is used. Any of these three levels

could intuitively be used to implement SER, but the whitepaper notably does not talk about modifications to any of those components. Instead, it says Ada has “optimizations to the SM and memory system specifically targeted at efficient thread reordering”.

I’m guessing SER only reorders threads within the SM. That reduces the scope of reordering, making the process faster. Each Ada SM can have 48 warps in flight, or 1536 threads. At the GPC level, a group of six SMs could have over 9000 threads active. Alternatively, reordering across the entire GPU would be another order of magnitude more difficult. Furthermore, reordering within a SM avoids having to synchronize threads across the entire GPU. All of the threads that could potentially exchange places need to save their live state before the reordering can happen, and doing a GPC-wide or GPU-wide barrier would be quite costly. Finally, SM-private resources could be used to speed up reordering. Nvidia’s Ada Lovelace whitepaper doesn’t talk about L1 cache and shared memory splits, but the Ampere whitepaper says that graphics workloads get 64 KB of L1, 48 KB of shared memory, and 16 KB of reserved space. The 32-bit sort keys for all of the SM’s 1536 threads would fit within shared memory. Shared memory is a small software managed scratchpad local to a SM. Compared to the regular cache hierarchy, shared memory provides higher performance and could be used to help reordering within the SM.

Cache Optimizations

NVIDIA’s whitepaper also mentions “memory system optimizations”. While coherence hints would easily fit within the L1 cache or shared memory, the same doesn’t apply to vector register state. Ada has enough L2 to handle every SM saving its register contents, with capacity to spare.

GPU	SM/WGP Count	Vector Register	Total Vector	Cache Capacity

		Files Per SM/WGP	Register File Size	
Nvidia RTX 4070	46	4x 64 KB	11.7 MB	L2: 36 MB
Nvidia RTX 4070 Ti	60	4x 64 KB	15.3 MB	L2: 48 MB
Nvidia RTX 4090	128	4x 64 KB	32 MB	L2: 72 MB
AMD Radeon RX 6900 XT	40	4x 128 KB	20.4 MB	L2: 4 MB IC: 128 MB
AMD Radeon RX 7900 XTX	48	4x 192 KB	36.8 MB	L2: 6 MB IC: 96 MB

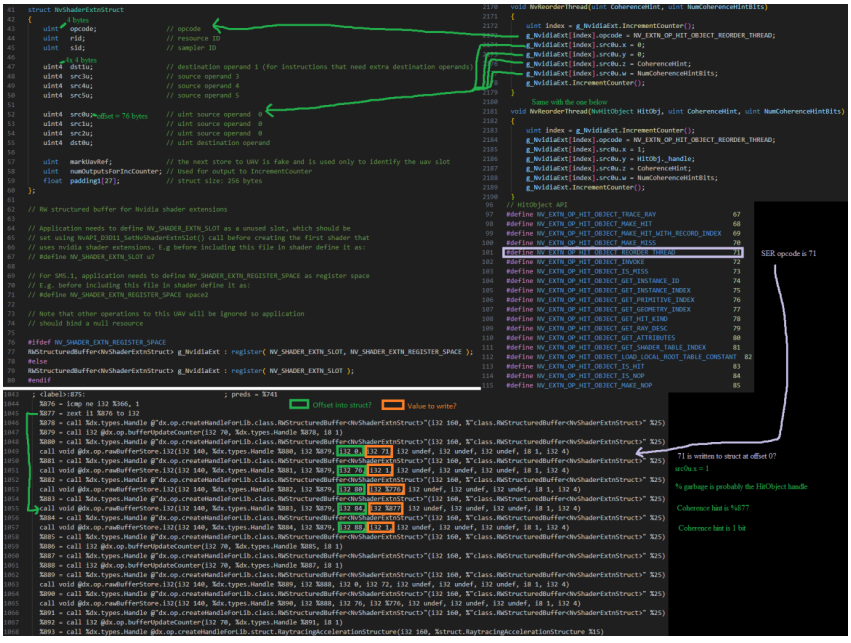
The L2 cache on both AMD and NVIDIA is the first write-back cache level on the vector memory hierarchy. Write-back means register state can be written to the cache, without having to propagate the writes on to the next level. We previously saw that Ada's L2 combines its large capacity with very good performance. With SER, these characteristics likely play a large role in making sure reordering can happen with low overhead.

AMD makes a different tradeoff. RDNA 2 and 3 use a very fast L2, but use another level of cache to provide massive caching capacity. AMD's L2 might not be large enough to absorb register file spills if they attempt reordering, unless each thread has very little live state data to contend with.

Evidence for SER's Use

Figuring out whether a game takes advantage of a particular feature can be difficult, particularly when profiling tools don't show shader assembly, or have counters that directly indicate the feature's being used. Nsight can show shaders in DXIL form though. DXIL stands for DirectX Intermediate Language. An intermediate language is called intermediate because it combines the worst aspects of high and low level languages to create an unreadable mess.

Comparing DXIL shows that the SER shader writes a set of values to a NvShaderExtnStruct structure. These writes are absent from the non-SER shader, where NvShaderExtnStruct doesn't appear at all. But NvShaderExtnStruct is used for other Nvidia-specific functions too. To get a better idea of whether it's SER related, I downloaded NVAPI R530. While I don't have any SER-capable Nvidia cards, I can look at NVAPI's header files along with the DXIL.



There's no guarantee on how a compiler will lay out a struct in memory. But the most intuitive way is that it lays it out the elements sequentially, with no padding as long as elements are aligned. "Opcode" is the first field in the struct, so it should be at offset 0. "src0u" should start 76 bytes in, because the fields before it in the struct occupy 76 bytes. The

x, y, z, and w components of “src0u” should be adjacent, putting them 76, 80, 84, and 88 bytes into the struct, respectively.

DXIL code calls rawBufferStore, probably writing the numerical value 71 to the struct at offset 0. So, it’s setting the opcode to 71, or NV_EXTN_OP_HIT_OBJECT_REORDER_THREAD. “src0u” has its x component set to 1, its y component set to an unknown value, z component set to a value that’s generated by checking whether some other unknown value is equal to 1, and its w component set to 1. All of that is consistent with the NvReorderThread function that takes a NvHitObject, a CoherenceHint, and the number of bits to check in the coherence hint. In this case, the coherence hint is a true or false value. That means sorting should be quite fast, and have low memory overhead.

From going through the DXIL, I’m convinced the DXIL shaders provided by SebaRTX do indicate SER’s being used in Cyberpunk 2077’s Overdrive mode.

Final Words

NVIDIA has invested heavily into raytracing over the past few generations. The company has no doubt poured significant resources into raytracing acceleration hardware, as well as upscaling technologies to cope with raytracing’s massive appetite for compute power.

But beyond requiring a lot more work than rasterization, raytracing has characteristics that make GPUs struggle to efficiently utilize their shader arrays. Shader execution reordering, alongside intersection test hardware, is yet another tool that lets Nvidia make raytracing go faster.

As with any technology, SER has limitations. Reordering incurs overhead, so developers have to be cautious when employing SER. While SER reduces divergence, raytracing still suffers from worse coherence than rasterization or other

compute workloads, so there's still plenty of progress to be made. As a result, I can't quite shake the feeling that the underlying SIMD nature of GPUs makes them fight an uphill battle when doing raytracing.

If you like our articles and journalism, and you want to support us in our endeavors, then consider heading over to our Patreon or our PayPal if you want to toss a few bucks our way. If you would like to talk with the Chips and Cheese staff and the people behind the scenes, then consider joining our Discord.

Credits

Thanks to SebaRTX for collecting the traces, as well as DXIL from the shaders.

Author

clamchowder

View all posts ✉

← Cyberpunk 2077's Path Tracing Update

ARM's Cortex A53: Tiny But Important →

3 thoughts on “Shader Execution Reordering: Nvidia Tackles Divergence”

Epliz says:

May 18, 2023 at 11:56 pm

Volta had “per thread program counter”. What happened to that? It isn’t useful for this case?

Reply

Dolda2000 says:

May 19, 2023 at 5:04 am

I could very well be wrong, but my understanding is that Volta’s per-thread PCs were primarily for enabling intra-wavefront locking possible, the idea being that if one set of threads waited for a lock that would be released by the some other set of threads in that same wavefront, that other set of threads could still make progress, but the sets would still execute divergently. That makes it solve a different problem than SER, whose purpose is just to decrease divergence, rather than improve thread independence on an architectural level.

I haven’t actually worked with this though, it’s just what I’ve gathered from whitepapers and stuff, so again I could be wrong.

Reply

Lukas Bergdoll says:

May 20, 2023 at 3:37 am

You are definitely onto something here. I remember seeing this talk at CppCon 2017 https://youtu.be/86seb-iZCnI?t=2054 and he talks about forward progress guarantees on the GPU. And them specifically designing Volta to support the C++ atomic programming model. His 2019 talk goes into more detail on that.

Reply

Leave a Reply

This site uses Akismet to reduce spam. Learn how your comment data is processed.