

General performance report

Note that this is to be considered a 'percentage scale' rather than a fixed numbers statistics.

This is also my own personal research and investigation through the render pipeline, any corrections/support from the team is requested and more than welcome.

Status	IN PRE-PRODUCTION
Owner	@Jad Deeb
Contributors	
Last Edited	02 Nov 2021

This is a single frame capture data statistics from Stress test level that can be accessed with the argument : -autotest AutoTest\frametime-perf_1600_stress_test.lua

In depth explanation of the following snapshots from PIX will be explained at a high level to understand where we stand performance wise on Xbox S.

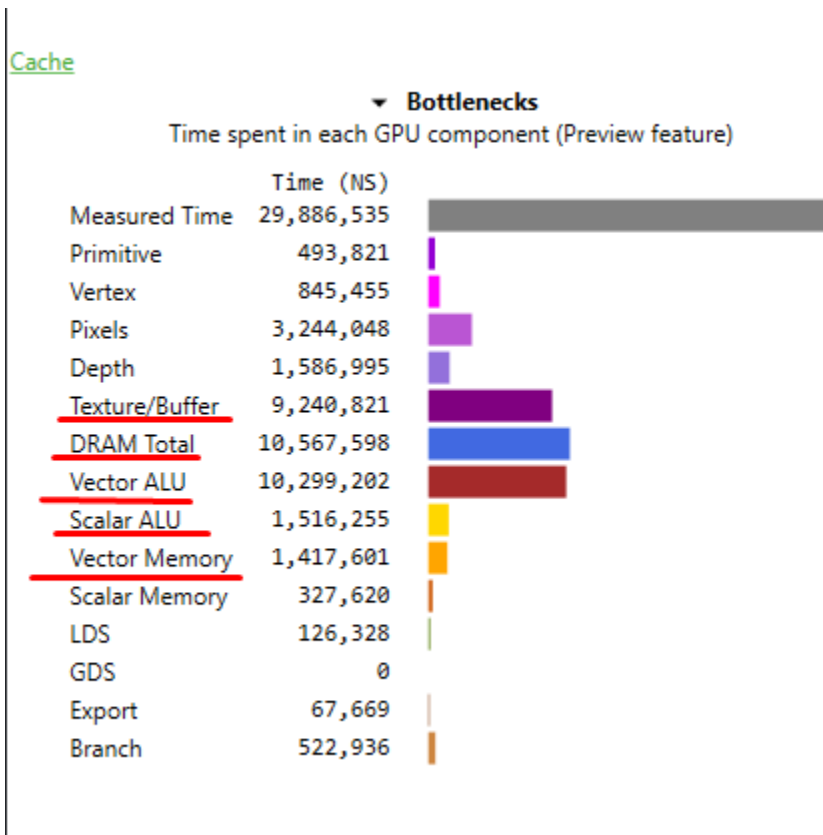
Bottlenecks :

General state :

Too many textures present at once in our memory.

Too many variables being read/generated in our memory.

Any new shaders that require Texture memory require render programmer support in order to plug it into the system.



More explanation:

From our main statistics in PIX, we can understand that we have a huge inconsistency in our rendering costs.

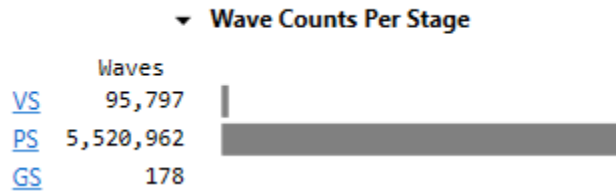
In which, on a very high level, the XBOX S is struggling in terms of Memory allocation. There's a great amount of textures present in our buffer (which is bad).

Thus amount of memory read and written is massive.

There's also a great amount of variables being utilized per frame(Vector ALU), in which in optimal situations Vector ALU and Scalar ALU should be within closer ranges for faster sailing.

This is more generally due to having many branches in a shader (Threads in same wave are attempting to read different functions).

Please note that : VectorALU ScalarALU refer respectively to Variables and Constants in a shader rather than the more used terminology of Vectors (3 scalars in a row or column) and Scalars (A single number such as 0, 1 or 1.4)



In general though, almost all the work/optimization is needed to be done on the Pixel Shader (PS).



(Please note that this is what I currently think is true and may be not 100% accurate)

Most notable stages that are taking the most amount of memory would be :

- Terrain generation
- Terrain generation for tiles that will cast shadows (Near Shadow Stage)
- Depth testing (DepthFirst stage)
- Creating the contours around every single unit (ContourOut)
- Screen space ambient occlusion generation.
- Occluding objects in the scene (What goes in front, what does not get rendered etc)
- Terrain stage (unsure of what this currently does)
- Grass rendering, which is using a scary amount of textures for an unknown reason.
- Rendering all the fx elements. (NFX Stage)
- Generating Fog around the far edges of the screen and applying it
- Bloom
- Generating the FOW in both Map and Screen and applying it.
- Generating and drawing UI

Important to note that the worst (lowest) wave occupancy is present in the occlusion and terrain stages. Grass Rendering stage as well.

More low level analysis of our current state (Not for the faint hearted !) :

General note is that we are largely TEX bound, locality of resources is quite poor and it's hard to hold everything in context at once.

Too many bound resources cause efficiency in caching to become much lower.

▼ L2 Cache ⓘ			
GL2C - Graphics Level 2 Cache			
	Requests	Percentage	Bytes
▼ Outgoing Requests to Memory			
▸ Reads	31,937,744	100.00%	3,979,368,128
▸ Write	42,759,324	100.00%	2,697,403,552
▼ L2 Efficiency			
Hits	71,228,309	60.98%	
Misses	45,573,414	39.02%	
▸ MDC Efficiency			
▸ Incoming Request Type			
▸ Incoming Request Clients	116,510,013		

▼ L1 Cache ⓘ			
GL1C - Graphics Level 1 Cache			
	Requests	Percentage	Bytes
▼ Outgoing Requests to L2			
▸ Reads	73,003,341	100.00%	9,191,992,608
▸ Writes	43,605,958	100.00%	2,745,396,448
▼ L1 Efficiency			
Hits	50,364,054	30.20%	
Misses	116,428,282	69.80%	
▼ Incoming Request Type			
▸ Reads	123,248,086	100.00%	15,611,003,808
▸ Writes	43,607,913	100.00%	2,745,666,816
Atomic w/o Return	1,302		
Atomic w/ Return	7,377		
▸ Incoming Request Clients			

A few notes is that CPU optimizations are due, in particular the way the textures are bound.

On the GPU Side, reducing the amount of General purpose cases in the shaders, which is present almost everywhere.

This comes back to branching many paths that are never taken.

Optimizing the pixel shaders will potentially yield most gains.

Render stages analysis :

First thing we are greeted by is PIX screaming that there's way too many Texture Addresses (TA) going on, this means too many textures being read /written.

Some of these issues are VMEM bound, which means that most of the textures we are requesting/sending are Vector ALU (Variables).

The xbox simply does not have that much capacity (waves/warps) to read/write all of that, thus the error (Vector ALU is starved for instructions)

This keeps snowballing causing more waiting and bottlenecks.

All Stages Analysis i

Combined analysis of shaders in all hardware stages

Mapping of hardware to Direct3D stage:

CS (Compute shader)	CS
ES (Export shader)	n/a - hardware only vertex-processing portion of geometry shader
GS (Geometry shader)	GS
HS (Hull shader)	HS
LS (Local shader)	n/a - hardware only tessellation stage
PS (Pixel shader)	PS
VS (Vertex shader)	VS

▼ Warnings

- ⚠ VMEM instructions are stalled 3.43% due to full TA command FIFO (parts of the shader are VMEM issue bound).
- ⚠ VMEM instructions are stalled 6.67% due to full TA address/data FIFO (parts of the shader are VMEM issue bound).
It might be helpful to limit occupancy if stalls are > 15-20%.
- ⚠ Vector ALU is starved for instructions 43.60%. This could mean there's not enough waves to select Vector ALU instructions from.
Reorganizing shader may lead to faster overall execution. Also it could mean that the shader doesn't have enough Vector ALU instructions.
- ⚠ VMEM latency is 631 clocks which is high, poor access locality or low number of waves might be the reason.
- ⚠ Poor issue rate of 1.69 clocks per SIMD32 per instruction.
- ⚠ Average vector ALU instructions cost is 1.16 indicating non-full-rate instructions.

▼ Wave Counts Per Stage

	Waves	
VS	101,906	
PS	5,571,972	
GS	181	

Going back to the stalling, most of the amount of time is being lost/spent on waiting for waves/warps to be empty.

▼ Top bottlenecks i

Time spent issuing each instruction type per-wave

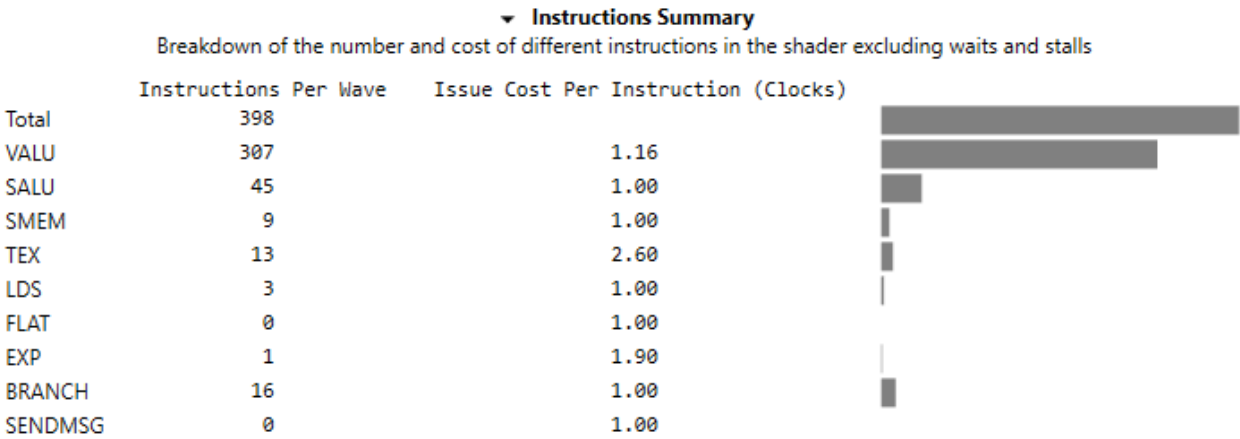
The biggest number indicates what the shader spend most of its time doing.

	Clocks	%	
Total	10,269.50	100.00%	
Waits	4,522.51	44.04%	
VMEM waits	4,296.25	41.84%	
VALU	1,920.73	18.70%	
VMEM	852.42	8.30%	
LGKM waits	156.11	1.52%	
EXP waits	70.15	0.68%	
SALU+SMEM	63.78	0.62%	
SALU	45.40	0.44%	
SMEM	9.32	0.09%	
LDS	4.90	0.05%	
EXP+GDS	3.32	0.03%	

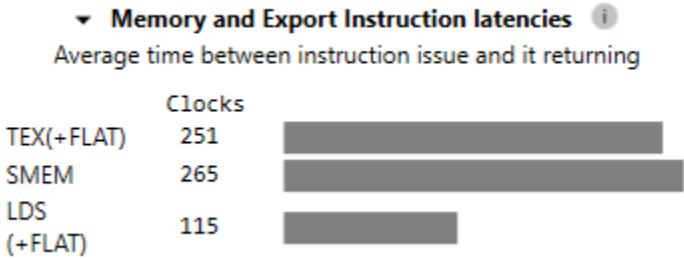
Going back to having too many variables, even though a single variable present costs 1.16 Clocks (unit of speed relating to each clock tick in the GPU), each wave is holding 307 instructions,

that's 356.12 clocks being spent just for variables each in each wave. This contrasts strongly with the only 45 constant instructions being sent per wave.

Important to note that in each wave 13 texture instructions are being occupied.



In the end, texture access (TEX) and Scalar Memory (SMEM) are being delayed in their input/outputs due to the previous explained latencies.



Texture fetching :

We are simply trying to read/write too many textures.

▼ Texture Addressing (TA) unit requests ⓘ

The breakdown of TA requests generated by shader instructions

Inst - the number of instructions of a particular type processed by Texture Addressing (TA) unit. Instructions coming from Wave64 shaders can be counted as one or two.

TA Req: The number of requests TA created from all processed instructions of a given type.

TA Per Inst: The average number of request TA creates per instruction type.

TCP Req: The number of TCP (aka L0) cache requests generated from all TA requests.

TCP per TA Req: The average number of TCP cache requests generated per a TA request.

L1\$ Req: The number of requests outgoing to L1 cache.

Type	Inst	TA Req	TA Req Per Inst	TCP Req	TCP per TA Req	L1\$ Req
Buffer Read	10,643,761	16,625,753	1.56	38,779,722	2.33	4,451,090
Buffer Write	17,022	17,118	1.01	41,804	2.44	57,576
► Buffer Atomic	6,579	6,579	1.00	8,049	1.22	8,049
▼ Image Read	78,622,293	782,444,412	9.95	1,144,975,373	1.46	110,861,970
Image Load		16,616,625		28,242,161	1.70	14,184,351
Image Sample		765,827,787		1,116,733,212	1.46	96,677,619
Image Write	5,708,544	5,708,544	1.00	10,892,544	1.91	17,372,544

Lower Mips are not being sampled are not used as often (or at all).

A direct and straightforward way to debug this is to use change the effective texture size by using high MIP LOD Bias. If this greatly improves performance then it is further confirmed.

▼ Mip and Multisample Samples ⓘ

The percentage of texture samples read from each MIP and Multisample level

Note that tbuffer loads will appear as MIP 0.

Level	Fetches	%
0	2,124,754,414	74.23%
1	462,511,935	16.16%
2	141,667,816	4.95%
3	60,936,468	2.13%
4	42,071,836	1.47%
5	23,981,430	0.84%
6	5,382,172	0.19%
7	512,249	0.02%
8	185,482	0.01%
9	344,703	0.01%
10	0	0.00%
11	0	0.00%
12	0	0.00%
13	0	0.00%
14	0	0.00%

Our DRAM is trying to read/write quite a bit of memory.

▼ DRAM Bandwidth

Amount of memory read and written to main memory

	Bytes	MB/s	
DRAM Cached Read	3,828,891,296	122,493	
DRAM Non-cached Read	238,080	7	
DRAM Read Total	3,829,129,376	122,501	
DRAM Cached Write	2,515,666,208	80,481	
DRAM Non-cached Write	8,588,128	274	
DRAM Write Total	2,524,254,336	80,755	
DRAM Total	6,353,383,712	203,257	

General suggestions :

The screen is being fully rendered at all times, a small suggestion to save some pixel rendering time would be to occlude and not draw the pixels that are always behind the UI.



Suggestions on how that may be done :

- Leveraging the StencilBuffer that is already being used.
- Add to nearest depth plane and get pixel fragments to depth reject.
- Rendering the ScreenSpace UI first and using that as a stencil.