Xbox 360 Alpha vs. Final Hardware Performance

Introduction

How will your game perform when you move it from Xbox 360 alpha hardware to Xbox 360 final hardware? The true answer is complex and dependent on many factors, including the details of your engine and rendering strategy. We don't have final hardware yet, and there are often considerable differences between actual and expected performance. This paper will analyze the differences between the performance of alpha hardware and the expected performance of final hardware with respect to several key factors.

Notes

- This paper does not provide benchmarks for hardware other than alpha and final. If your engine is still PC based, it's up to you to find the performance numbers of your current hardware and compare them to the expected performance of final hardware.
- This paper does not provide benchmarks for the original alpha kit R300 graphics chip. It only discusses performance with respect to the R420 upgrade.

If you haven't ported your Xbox 360 title to the alpha kit, we suggest that you do so soon. The Xbox 360 XDK is a stable platform and hundreds of developers are using it successfully. There are several good reasons to move to the alpha XDK.

- If there is a bug that is specific to your engine's usage of the XDK, the odds of getting it fixed before launch are much better if we can identify it sooner.
- Our monthly ship schedule means a short turn around for critical bug fixes.
- There are significant DirectX differences and extensions that would be preferable to address earlier rather than later.
- Finally, the XDK has an extensive suite of emulation and performance tools that are not available on the PC but are critical to Xbox 360 development. It would be a mistake to not take advantage of these tools now, even if some of the numbers are skewed due to differences in alpha vs. final hardware.

The rest of this paper will cover detailed differences between the alpha and final CPUs, GPUs, and DirectX performance. To better illustrate these differences, the following key is used throughout the paper to show performance impact.

Performance Impact Key	
🟩 ↑↑	Much better
🟩 ↑	Better
–	About the same
🟧 ↓	Worse
🟥 ↓↓	Much worse

CPU Differences

This section explains the major performance-related differences between the CPUs and gives examples where appropriate.

Architecture

	Scale	Alpha	Final	Multiplier	Impact
Frequency	GHz	2	3	1.5	↑
Cores	cores	2	3	1.5	↑
Hardware threads per core	hardware threads	1	2	2.0	↑↑
Max instructions per cycle per core	instructions/ cycle	5	2	0.4	↓↓
Max instructions per second per core	MIPS	10000	7000	0.7	↓
Max floating-point ops per cycle per core	FLOPS/cycle	12	8	0.7	↓
Max floating-point operations per core	GFLOPS	24	28	1.2	↑
Max floating-point operations total	GFLOPS	48	84	1.8	↑↑
Supports out-of-order execution		yes	no		↓↓

There will be three cores in the final CPU that run at 3 GHz per core. Each final hardware core supports two hardware threads. Alpha hardware has two separate CPUs that run at 2 GHz each with no hardware threads.

Do not make the mistake of thinking that higher frequency and more cores will necessarily mean better performance. In some cases, the final cores will run code faster. In some cases, they will run code slower.

In current CPU architectures, there are often tradeoffs made in clock rate vs. complexity. The more complex the hardware, the smaller the maximum clock rate can be. Conversely, when hardware is designed to run at high clock speeds, simplicity is critical.

The alpha CPUs run at a lower clock rate than the final CPU cores, but each CPU can dispatch three instructions more per cycle than a final hardware cores. The alpha CPUs support out-of-order execution and the final CPU cores do not. This means that clock for clock, the alpha CPUs are probably faster running generic code, where "generic code" is defined as code that is non-streaming and is written for general non-platform-specific use.

The custom Xbox 360 compiler will help optimize for the in-order nature of the final hardware's CPU cores. Also, in-order architectures are inherently easier to benchmark and hand optimize for.

Register Space

	Scale	Alpha	Final	Multiplier	Impact
Integer registers per hardware thread	64-bit integers	32	32	1.0	–
FPU registers per hardware thread	doubles	32	32	1.0	–
Vector registers per hardware thread	float4s	32	128	4.0	↑↑
Register space per core	bytes	1024	5120	5.0	↑↑
Total architectural register space	bytes	2048	15360	7.5	↑↑

The integer and float register spaces are the same per thread and are fairly generous on both alpha and final hardware. Final hardware has four times as many vector registers per thread as alpha hardware, so we expect to see big wins in vector processing routines that make good use of the larger vector register space.

Final hardware has separate integer, float, and vector register spaces for each of its two hardware threads. Along with other hardware threading optimizations, this greatly improves the speed of multiple threads per core on final hardware.

Caches

	Scale	Alpha	Final	Multiplier	Impact
Cache line size	bytes	128	128	1.0	–
L1 instruction cache size (per core)	KB	64	32	0.5	↓
L1 instruction cache associativity	associativity	1-way	2-way	2.0	↑
L1 instruction cache hit latency	clocks	~3	4	0.8	↓
L1 data cache size (per core)	KB	32	32	1.0	–
L1 data cache hit latency	clocks	~3	5	0.6	🟧
L1 data cache associativity	associativity	2-way	4-way	2.0	↑↑
L2 cache size	KB	512 (per core)	1024 (shared)	1.0	↑
L2 cache associativity	associativity	8-way	8-way	1.0	–
L2 hit latency	clocks	11	39	0.3	↓↓

Each alpha CPU has a 1-way 64-KB instruction cache and a 2-way 32-KB data cache. Final hardware has 32 KB for both on each core, but the associativity is better (2-way instruction and 4-way data). For both alpha and final hardware, a good percentage of the latency of an L1 hit is usually covered up by the pipeline depth and is usually not an issue. We suspect that final hardware will have slightly better L1 cache performance.

Each alpha CPU has an 8-way 512-KB L2 cache while the final CPU cores share an 8-way 1024-KB unified L2. The unified cache means that the cores will be able to share L2 data, and the entire 1024 KB will be available to each core. The net effects should mean less L2 cache misses overall.

However, the latency differences for an L2 hit in terms of cycles are larger on final hardware (~39 cycles on final vs. ~11 cycles on alpha). We expect that this will cause many routines to run slower on final hardware. However, the larger shared L2 should significantly reduce L2 cache misses.

Memory

	Scale	Alpha	Final	Multiplier	Impact
Memory latency	clocks	~205	~525	0.4	↓↓
Main memory bandwidth	GB/s	16 (8 GB/s per core)	10.8R+10.8W (shared)	1.4	↑

Final hardware has higher CPU-to-main-memory bandwidth of 10.8 GB/s read + 10.8 GB/s write (shared between all 3 cores) vs. 8 GB/s read/write (per CPU) on alpha hardware. However, only ~6.5 GB/s read/write of the final hardware's bandwidth can be read through the L2 cache due to limitations of the L2 RC machines. Non-temporal writes must be used to utilize the remaining bandwidth. The **Xbox 360 CPU Caches** white paper has detailed information and explanations.

While the total bandwidth available is larger, this bandwidth must be shared with the GPU due to the unified memory architecture. However, the final hardware's CPU has a higher priority for memory access than the GPU, so we don't expect the sharing to affect CPU performance.

The latency to fetch from main memory is considerably larger on final hardware (~525 cycles on final vs. ~205 cycles on alpha). We expect that this latency will be a significant factor and cause generic code to run slower on final hardware. This is one reason why cache optimization is so critical on final hardware for memory-intensive routines. Utilizing hardware threads to hide stalls is also a good technique to employ on final hardware.

VPUs

	Scale	Alpha	Final	Multiplier	Impact
fmad latency	cycles	6	10	0.6	🟥 ↓↓
Dot product instruction in hardware		no	yes		🟩 ↑↑
D3D vector pack / unpack in hardware		no	yes		🟩 ↑↑

The vector processing units (VPUs) on final hardware are much better than on alpha. Not only is the register space larger, but special instructions for dot products and D3D vector packing and unpacking are available on final hardware. Note that the latency for most VPU instructions is larger on final hardware (10 cycles on final vs. 6 cycles on alpha), so code scheduling will be critical.

CPU Summary and Recommendations

We recommend that hardware threading be used. We also recommend that cache control instructions be used where possible. It is also important to maximize the coherency of your memory accesses as much as possible. These optimizations should help avoid the latency issues on final hardware.

We understand that in the short term many of you will be unable to completely re-architect your engines to take advantage of threading and cache control. However, several key routines such as skinning, ray tracing, sound, shadow volume extrusion, and so on, can be offloaded to another core. Also, since many of these routines are streaming, the memory access can be controlled and the benefits of pre-fetching, doing non-cached stores, and maintaining cache coherency can be large wins. In the worst case, where very little of an engine is multi-threaded, the large, shared L2 cache, which will primarily be used for the one thread, will help boost performance.

If you are able to offload major systems such as rendering, AI, visibility processing, physics, and so on to another core or hardware thread, even bigger wins can be achieved.

The final CPUs are faster, but they suffer from more stalls due to architectural differences and longer cache and main memory latencies. We suspect that core for core, generic code will run slightly slower on final hardware—perhaps 25 percent slower. However, engines that are optimized for Xbox 360 have the ability to run perhaps twice as fast on final hardware vs. alpha hardware.

GPU Differences

The differences between the alpha kit's R420 GPU and the final hardware's GPU are significant.

Architecture

	Scale	Alpha (R420)	Final	Multiplier	Impact
Frequency	MHz	500	500	1.0	–
EDRAM	MB	0	10		↑↑
Primary Software API		Direct3D 9	Direct3D 9+		↑↑
Shader model		VS 2.0 / PS 2.0+	VS 3.0+ / PS 3.0+	1.0	↑↑
Xbox 360 extensions		no	yes		↑↑
Total programmable floating-point ops	GFLOPS	88	216	2.5	↑↑
State pipelining		Pipelines affected	8 simultaneous contexts		↑↑
Shader precision	bits	24-bit float	32-bit float	1.3	↑

While both the final GPU and the alpha GPU run at 500 MHz, the similarities end there. The final GPU is simply capable of doing much more per clock and is less bandwidth-bound than the alpha's R420.

The R420 has 6 vertex processing pipes and 16 pixel processing pipes. The vertex pipes work on a single vertex at a time, and the pixel pipes work on four 2x2 pixel quads at a time. Due to the linear nature of the pipes on the R420, it is also possible for the R420 to be bottle-necked in either vertex processing or pixel processing. Full-screen pixel processing effects, for example, will leave the vertex pipe mostly idle.

The final GPU is a large change from the pixel and vertex pipe concept of the R420. The best place to learn the details of the new GPU is the **Xbox 360 GPU Overview** white paper. Briefly, the final GPU schedules many simultaneous threads that work on large vectors (64 entries deep) of data at a time. It also uses EDRAM to separate frame-buffer bandwidth from texture and vertex fetch bandwidth, which is a huge savings. Due to the unified pixel and vertex shader model, there should be less wasted processing power due to imbalances in pixel and vertex processing. For example, the GPU can be almost completely utilized for pixel shading when doing post-processing effects.

The R420 supports shader model 2.0, while final hardware supports shader model 3.0 with some useful extensions such as memory export, fetch offsets, separate blend states for multiple render targets, compressed high dynamic range formats, programmable indexing, and so on. We expect that the addition of these extensions and the upgrade in shader model support will greatly improve both the look and speed of most rendering solutions.

The R420 supports static branching for vertex shaders only. Other forms of branching must be done with predication (executing all possible branches and selecting the correct result). The final GPU supports static and dynamic branching for both vertex and pixel shaders. On the final GPU, dynamic branching can share the same performance characteristics as static branching if all 64 pixels or vertices in a thread's working group take the same branch or iteration. If you are using branching in your pixel shaders, expect to see a performance gain.

On the R420, when a state change occurs, the portion of the pipe affected by the change must be flushed. The final hardware's GPU can maintain 8 separate contexts, so we expect that state change performance will be better than the 420 with final hardware.

Finally, GPU calculations done on the final hardware have 8 more bits of precision, which will help keep colors crisper and calculations more accurate.

Vertex Processing

	Scale	Alpha (R420)	Final	Multiplier	Impact
Max triangle rate	megatris/s	500	500	1.0	–
Max vertex rate	megaverts/s	500	500	1.0	–
Vertex shader ALU performance	ops/cycle	6	48 (shared with PS)	8.0	↑↑
Max vertex shader instructions	instructions	256	4096	16.0	↑
Max vertex shader float constants	float4s	256	256	1.0	–
Max vertex shader Boolean constants	Booleans	16	128	8	↑↑
Max vertex shader loop constants	integers	1	16	16	↑↑
Max vertex shader temp Registers	float4s	16	64	4.0	↑
Max vertex/index fetch bandwidth	GB/s	32 (shared with FB+PS)	22.4 (shared with CPUs+PS)	0.7	↑
Post-transform cache size	vertices	16	16	1.0	–

While the max vertex and triangle rate of both cards are the same, final hardware will have more bandwidth available for fetching given that frame buffer rendering does not share the same bus. For example, while the R420 has 32 GB per second of vertex and index fetch bandwidth, it must share this with both the front buffer which is a significant draw from bandwidth. The final hardware's use of EDRAM should alleviate this concern.

Vertex shader programs on final hardware can be larger and access more temporary registers. They also are unified with the pixel shaders, so texture sampling can be done in the vertex shader.

The differences in ALU processing for vertex transform and lighting are significant, with the final GPU having 8 times more ALU power available for vertex operations.

In general, we expect the final GPU to exceed the vertex processing power of the R420.

Pixel Processing

	Scale	Alpha (R420)	Final	Multiplier	Impact
Max texture rate	gigasamples/s	8	8	1.0	–
Pixel shader ALU performance	ops/cycle	16	48 (shared with VS)	3.0	↑↑
Interpolated pixel shader Inputs	float4s	10	16	1.6	↑
Max pixel shader instructions	instructions	512	4096	8.0	↑
Max pixel shader Constants	float4s	64	256	4.0	↑
Max pixel shader Boolean constants	Booleans	0	128		↑↑
Max pixel shader loop constants	integers	0	16		↑↑
Max pixel shader temp registers	float4s	32	64	2.0	↑
Max texture fetch bandwidth	GB/s	32 (shared with fill+VS)	22.4 (shared with CPUs+VS)	0.7	↑

While the maximum texture fetch rate is the same, again we expect that final hardware will exceed the R420 in this benchmark due to the alleviation of frame-buffer bandwidth sharing by using EDRAM.

Pixel shaders can be longer on final hardware; they can have a larger register space for both temporaries and constants, and they can have significant improvements in available ALU processing power.

In general, we expect the final GPU to again exceed the R420 in pixel processing power.

Fill

	Scale	Alpha (R420)	Final	Multiplier	Impact
Max pixel fill rate	gigapixels/s	8	4	0.5	↓
Max Z-only pixel Fill Rate	gigapixels/s	16	8	0.5	↓
Max MSAA fill rate	gigasamples/s	16	16	1.0	–
Max fill bandwidth	GB/s	32 (shared with VS+PS)	256	8.0	↑↑
Bandwidth utilization – 32 bpp color only	% utilization of total	100.00%	6.25%	16.0	↑↑
Bandwidth utilization – 32 bpp color-Z	% utilization of total	300.00%	18.75%	16.0	↑↑
Bandwidth utilization – 32 bpp color-Z-alpha	% utilization of total	400.00%	25.00%	16.0	↑↑
Bandwidth utilization – 32 bpp color-Z-alpha 2x MSAA	% utilization of total	800.00%	50.00%	16.0	↑↑
Bandwidth utilization – 32 bpp color-Z-alpha 4x MSAA	% utilization of total	800.00%	100.00%	8.0	↑↑

The increase in bandwidth for fill due to EDRAM will speed up bandwidth-bound operations significantly.

While the R420 technically can do twice the number of quads per cycle, it quickly becomes frame-buffer bandwidth-bound in even the simplest case. For example, a simple color write with Z-testing requires 300% of the available frame-buffer bandwidth on the alpha GPU. The final GPU never becomes frame-buffer bandwidth-bound.

We expect that the available bandwidth due to EDRAM will significantly increase realizable fill speed, especially when Z-testing, alpha-blending, and MSAA are used.

GPU Summary and Recommendations

While the bandwidth savings from EDRAM are significant, using EDRAM often requires reworking the rendering portion of your engine. It is recommended that you consider the impact and benefits of using EDRAM early.

Given the 64-entry working set nature of the final GPU, we recommend that you shade as many vertices or pixels per draw call as possible.

Overall, we expect the final GPU to be faster in most respects over the R420. The additional ALU processing power, bandwidth, and features should be a large improvement. Depending on your rendering style and usage of the hardware, we expect this improvement to be anywhere from 25 percent for basic lit polygon rendering to several hundred percent increases if heavy alpha blending, MSAA, ALU operations, or static and coherent dynamic branching are used.

Direct X Differences

	Scale	Alpha (R420)	Final	Multiplier	Impact
CPU dual driver Overhead		Significant	None		↑↑
Dynamic resource bandwidth	GB/s	1.05 (AGP)	22.4 GB/s (shared)	21.3	↑↑
Resolve and format conversion		Render to texture	Hardware		↑↑

There are several key changes to DirectX that will appear with the move to final hardware.

Currently, Direct3D is maintaining a mixed state for both the R420 and the final hardware. The CPU and memory overhead involved with DirectX will be significantly better on final hardware when only a single state is maintained.

On the alpha kit, static resources are stored in video RAM for fast access by the GPU. Dynamic resources on the alpha kit are stored in AGP memory and accessed by the GPU at 4x AGP speed, or around 1.05 GB/s. Final hardware has a unified memory architecture, so both static and dynamic resources can be accessed by the GPU at full speed, or 22.4 GB/s.

The HLSL compiler currently shipping with the XDK is not optimized for final hardware. Its optimization capabilities for final hardware, especially with respect to static and dynamic branching, will improve significantly as HLSL compiler optimizations are introduced.

Finally, a significant amount of bandwidth and pixel processing power are being consumed by Direct3D to emulate EDRAM on the R420. Specifically, on the R420, every resolve results in a render to texture, and format conversion is done in a pixel shader. On the final hardware, this emulation will be history.

Audio Differences

	Scale	Alpha	Final	Multiplier	Impact
Hardware XMA decoder		no	yes		↑↑

Because the primary audio hardware component (a multichannel XMA decoder) is not available on alpha hardware, there are significant differences in performance. For more details on audio feature schedules, see the **Xbox 360 Audio Roadmap** white paper.

The presence of an XMA decoder, along with ongoing CPU optimizations for the XAudio low-level audio library, will lead to significant reductions in memory footprint and relatively equivalent CPU utilization when compared with the use of PCM wave data.

Content creators should utilize the XMA encoder already available in the XDK to familiarize themselves with the format and appropriate quality/compression trade-offs. Programmers should integrate the XMA decoder (command-line or library) into their build process. For the alpha time frame, titles should likely continue to use PCM almost exclusively, as XMA software decoding is prohibitively expensive for multiple streams. On final hardware, XMA should typically be used for all sounds. Mixing and digital signal processing will likely see improved performance once they are on native final hardware.

Conclusion

Our first recommendation: profile your game. You need to know where your bottlenecks are before you can put the data provided in this paper to good use.

If you are CPU bound, optimizations done to your code and algorithms will most likely provide benefits on both alpha and final hardware. We would not begin assembly or lower-level optimizations until algorithms and techniques are optimized. Remember, for non-optimized, generic, single-threaded code, we expect the performance to be slightly worse on final hardware. For such cases, consider using cache and memory-control instructions and multithreading to help out.

If you are GPU bound, optimize algorithms and techniques first before beginning micro-optimizations. An unused clear of the front buffer is a performance drain on both platforms, as is a poorly implemented visibility system. We expect the performance of the GPU to be better on final hardware, but its still worth finding the bottlenecks on the alpha GPU to verify that the new GPU will indeed help out and by what margin.

Finally, while this paper represents our best guess on what the performance differences will be, it is realistic to expect that some of these numbers will change before final hardware is manufactured. It is also expected that other performance bottlenecks and critical characteristics may arise. When changes occur, we will be sure to keep you updated.

Complete Comparison Chart

Performance Impact Key	
↑↑	Much better
↑	Better
–	About the same
↓	Worse
↓↓	Much worse

		Scale	Alpha (R420)	Final	Multiplier	Impact
CPU						
Architecture	Frequency	GHz	2	3	1.5	↑
	Cores	cores	2	3	1.5	↑
	Hardware threads per core	hardware threads	1	2	2.0	↑↑
	Max instructions per cycle per core	instructions/ cycle	5	2	0.4	↓↓
	Max instructions per second per core	MIPS	10000	7000	0.7	↓
	Max floating-point operations per cycle per core	FLOPS/cycle	12	8	0.7	↓
	Max floating-point operations per core	GFLOPS	24	28	1.2	↑
	Max floating-point operations total	GFLOPS	48	84	1.8	↑↑
	Supports out-of-order execution		yes	no		↓↓
Registers	Integer registers per hardware thread	64-bit integers	32	32	1.0	–
	FPU registers per hardware thread	doubles	32	32	1.0	–
	Vector registers per	float4s	32	128	4.0	↑↑

	hardware thread					🟩
	Register space per core	bytes	1024	5120	5.0	🟩 ↑↑
	Total architectural register space	bytes	2048	15360	7.5	🟩 ↑↑
Caches	Cache line size	bytes	128	128	1.0	–
	L1 instruction cache size (per core)	KB	64	32	0.5	🟧 ↓
	L1 instruction cache associativity	associativity	1-way	2-way	2.0	🟩 ↑
	L1 instruction cache hit latency	clocks	~3	4	0.8	🟧 ↓
	L1 data cache size (per core)	KB	32	32	1.0	–
	L1 data cache hit latency	clocks	~3	5	0.6	🟧
	L1 data cache associativity	associativity	2-way	4-way	2.0	🟩 ↑↑
	L2 cache size	KB	512 (per core)	1024 (shared)	1.0	🟩 ↑
	L2 cache associativity	associativity	8-way	8-way	1.0	–
	L2 hit latency	clocks	11	39	0.3	🟥 ↓↓
Memory	Memory latency	clocks	~205	~525	0.4	🟥 ↓↓
	Main memory bandwidth	GB/s	16 (8 GB/s per core)	10.8R+10.8W (shared)	1.4	🟩 ↑
VPU	fmad latency	cycles	6	10	0.6	🟥 ↓↓
	Dot product instruction in hardware		no	yes		🟩 ↑↑
	D3D vector pack/unpack in hardware		no	yes		🟩 ↑↑
GPU						
Architecture	Frequency	MHz	500	500	1.0	–
	EDRAM	MB	0	10		🟩 ↑↑
	Primary software API		Direct3D 9	Direct3D 9+		🟩 ↑↑
	Shader model		VS 2.0 / PS 2.0+	VS 3.0+ / PS 3.0+	1.0	🟩 ↑↑
	Xbox 360 Extensions		no	yes		🟩 ↑↑
	Total programmable floating-point ops	GFLOPS	88	216	2.5	🟩 ↑↑
	State pipelining		Pipelines affected	8 simultaneous contexts		🟩 ↑↑
	Shader precision	bits	24-bit float	32-bit float	1.3	🟩 ↑
Vertex	Max triangle rate	megatris/s	500	500	1.0	–
	Max vertex rate	megaverts/s	500	500	1.0	–
	Vertex shader ALU performance	ops/cycle	6	48 (shared with PS)	8.0	🟩 ↑↑
	Max vertex shader instructions	instructions	256	4096	16.0	🟩 ↑
	Max vertex shader float constants	float4s	256	256	1.0	–
	Max vertex shader Boolean constants	Booleans	16	128	8	🟩 ↑↑
	Max vertex shader loop constants	integers	1	16	16	🟩 ↑↑
	Max vertex shader temp registers	float4s	16	64	4.0	🟩 ↑
	Max vertex/index fetch bandwidth	GB/s	32 (shared with FB+PS)	22.4 (shared with CPUs+PS)	0.7	🟩 ↑

	Post transform cache size	vertices	16	16	1.0	–
Pixel	Max texture rate	gigasamples/s	8	8	1.0	–
	Pixel shader ALU performance	ops/cycle	16	48 (shared with VS)	3.0	↑↑
	Interpolated pixel shader inputs	float4s	10	16	1.6	↑
	Max pixel shader instructions	instructions	512	4096	8.0	↑
	Max pixel shader float constants	float4s	64	256	4.0	↑
	Max pixel shader Boolean constants	Booleans	0	128		↑↑
	Max pixel shader integer constants	integers	0	16		↑↑
	Max pixel shader temp registers	float4s	32	64	2.0	↑
	Max texture fetch bandwidth	GB/s	32 (shared with fill+VS)	22.4 (shared with CPUs+VS)	0.7	↑
Fill	Max pixel fill rate	gigapixels/s	8	4	0.5	↓
	Max Z-only pixel fill rate	gigapixels/s	16	8	0.5	↓
	Max MSAA fill Rate	gigasamples/s	16	16	1.0	–
	Max fill bandwidth	GB/s	32 (shared with VS+PS)	256	8.0	↑↑
	Bandwidth utilization – 32 bpp color only	% utilization of total	100.00%	6.25%	16.0	↑↑
	Bandwidth utilization – 32 bpp color-Z	% utilization of total	300.00%	18.75%	16.0	↑↑
	Bandwidth utilization – 32 bpp color-Z-alpha	% utilization of total	400.00%	25.00%	16.0	↑↑
	Bandwidth utilization – 32 bpp color-Z-alpha 2x MSAA	% utilization of total	800.00%	50.00%	16.0	↑↑
	Bandwidth utilization – 32 bpp color-Z-alpha 4x MSAA	% utilization of total	800.00%	100.00%	8.0	↑↑
Direct X						
	CPU dual driver overhead		Significant	None		↑↑
	Dynamic resource bandwidth	GB/s	1.05 (AGP)	22.4 GB/s (shared)	21.3	↑↑
	Resolve and format conversion		Render to texture	Hardware		↑↑
Audio						
Overall	Hardware XMA decode		no	yes		↑↑