Scaling of 3D Game Engine Workloads on Modern Multi-GPU Systems

Jordi Roca Monfort
Computer Architecture Department (UPC)
jroca@ac.upc.edu

Mark Grossman
AMD
mark.grossman@amd.com

Abstract

This work supposes a first attempt to characterize the 3D game workload running on commodity multi-GPU systems. Depending on the rendering workload balance mode used, the intra and inter-frame dependencies due to render-to-texture require a number of synchronizations that can significantly impact the scalability with multiple GPUs. In this paper, a proprietary analytical tool called EMPATHY has been used to evaluate, for a set popular DX9 games, the performance of both classic split frame and alternate frame rendering modes as well as combined modes supporting more than 4 GPUs. We have also evaluated the application of the early copy and concurrent update techniques together as alternative to delayed surface copy of render-to-texture surfaces, showing a 48% percent improvement for some workloads.

CR Categories: I.3.1 [Computer Graphics]: Hardware architecture—Graphics processors; C.1.4 [Processor Architectures]: Parallel Architectures—Distributed architectures;

1 Introduction

Nowadays, the enthusiast gaming segment demands antialiased high-resolution and outstanding image quality at higher frame rates, and high-end graphics products targeted for this market segment provide high pixel rate and memory bandwidth. However, over the last years, the use of multiple-GPU based solutions has become very popular due to better scalability and upgrade cost. This paper shows a new characterization of 3D games executed in multi-GPU configurations that helps to understand the key points that have influenced such parallel graphics systems, setting the groundwork for new specific driver optimizations.

Multi-GPU systems try to make the most of the overall available fill-rate and memory bandwidth by distributing the rendering workload among the different GPUs. NVIDIA's SLI [Relations 2005] and AMD's Crossfire [Persson 2005] offer limited multi-GPU solutions, allowing nowadays to interconnect up to four GPUs using special high-end motherboards. The graphics parts themselves also have dedicated hardware to efficiently reassemble partial renderings into the final sequence of displayed images.

As depicted in Figure 1, the GPU hardware accelerates the 3D rendering task processing rendering commands from the HAL (architecture-dependent driver), originated at the 3D engine part of the graphics application and processed down through the graphics API. The scalability of multi-GPU systems basically depends on 1) the ability of the driver to balance the render workload among the available GPUs, and 2) the data dependencies introduced by rendering commands that serialize the generation of the displayed image.

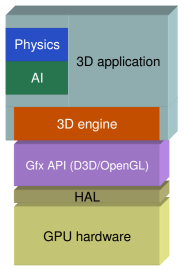


Figure 1: *3D application execution stack.*

Two primary rendering modes are used for load balancing. In Split Frame Rendering (SFR), each GPU renders a disjoint portion of the screen frame. The portions can be either assigned dynamically based on the previous frame workload history or, in case of tile-based rendering, partitioning the screen in a checkerboard fashion. Any of these alternatives requires sending all the geometry to every GPU as long as the screen-space area covered by the 2D projected triangles cannot be determined prior to the geometry object-to-screen space transformation, currently performed in GPU hardware at a lower cost than on the CPU. Consequently, as we increase the number of GPUs, the geometry workload becomes the bottleneck, making SFR a non-scalable solution.

In Alternate Frame Rendering (AFR), GPUs work in parallel on different consecutively assigned frames. This mode naturally balances the rendering workload as long as consecutive frames have roughly similar rendering workload. AFR scales the geometry processing workload since only the corresponding frame's geometry is sent to each GPU. One factor limits AFR scalability in interactive graphics: although frame generation rate increases, since each frame is processed by a single GPU, the user-input to result-on-screen latency is the same as in a single GPU, which can produce a noticeable delay.

The SFR-AFR combined modes use clustered GPUs to render individual frames in AFR mode, while GPUs within a cluster render a frame in SFR. Combined modes overcome the practical limitation with the pure classic modes due to the lack of scalability and decreased interactivity. In this paper we present simple SFR and AFR performance models to evaluate pure and combined load balance modes. Both models compute a penalization cost for synchronizing Render Target Texture surfaces (RTTs) through a simplified GPU interconnection model. The whole evaluation framework has been implemented extending a proprietary analysis tool, that we have called EMPATHY.

The effectiveness of a load balanced rendering mode highly depends on how the 3D engine manages rendered target texture (RTT) surfaces. Their contents are rendered on the fly based on the frame's scene information. Environment reflections on water is one example. Strictly implemented, we would need to synchronize every updated surfaced to every GPU in order to keep the data consistent. But in reality, we only need to update surfaces in order to respect data dependencies between frames, and only if such dependencies cross GPU boundaries: the RTT is rendered and later read by different GPUs. Intra and inter-frame dependencies of RTT surfaces is the key topic behind the analysis made in this paper, and will be widely covered.

In this paper, we also compare the cost of several data synchronization alternatives. The simplest way is copying just before a rendered texture surfaces is read. This on-demand copy always incurs the cost of data transmission cycles. We want to evaluate two different alternatives: the first places the start of the surface copy right after the last render of the surface, anticipating that the gap between the last render and the first use will partially or completely hide the copy cycles. The second alternative, which we call concurrent update, is even bolder and involves keeping each surface updated with the latest pixel values by transmitting all colorbuffer writes through the interconnection bus, updating each GPU's surface copy. The advantage of this alternative is that we expect the pixel shading cost in many cases to be dominant enough to hide the cost of the remote updates.

The remainder of this paper is organized as follows. Section 2 describes the analysis tool and the game workload list. Section 3 presents the important concepts behind multi-GPU workload execution as rendering modes or synchronization of rendered texture surfaces (RTTs). In Section 4, our characterization based on RTT surface usage analysis will be presented, setting the base for the multi-GPU evaluation using the performance models explained in Section 5. In Section 7 we talk about some previous work on multi-GPU systems. Finally, the results of our experimentations and conclusions will be summarized respectively in Sections 6 and 8.

2 Experimentation Framework

2.1 The analysis tool

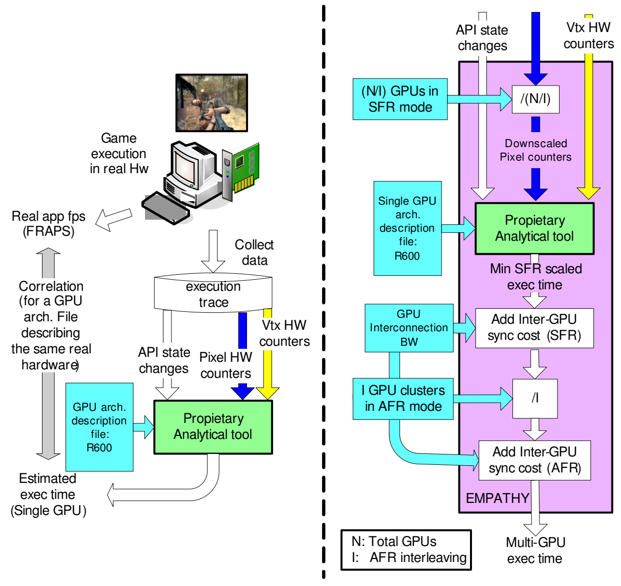

Game execution in real Hw
Real app fps (FRAPS)
Collect data
Correlation (for a GPU arch. File describing the same real hardware)
execution trace
API state changes
Pixel HW counters
Vtx HW counters
GPU arch. description file: R600
Propietary Analytical tool
Estimated exec time (Single GPU)
API state changes
Vtx HW counters
(N/I) GPUs in SFR mode
/(N/I)
Downscaled Pixel counters
Single GPU arch. description file: R600
Propietary Analytical tool
Min SFR scaled exec time
GPU Interconnection BW
Add Inter-GPU sync cost (SFR)
I GPU clusters in AFR mode
/I
Add Inter-GPU sync cost (AFR)
EMPATHY
N: Total GPUs
I: AFR interleaving
Multi-GPU exec time

Figure 2: *EMPATHY analysis tool.*

To evaluate multi-GPU performance in this paper, we have extended a proprietary analysis framework that estimates, for a set of games, the performance of different single GPU architectural configurations. This analysis tool, unlike many of the available game performance analysis tools [NVIDIA 2009a] that profile at run time, takes as input a trace file generated with collected data from the game execution in real hardware: API state changes as well as surface usage information, and vectors of performance counter values. The analytical model computes an approximation to the execution time for the evaluated architecture based on the gathered data and the GPU architecture description file, as shown in Figure 2 (left). For the present work, a description of a modern AMD R600 GPU architecture (Radeon HD 2900 XT) was used. All the gathered information is fairly orthogonal to any concrete architectural parameter (e.g. processed vertices, z-tested fragments) so we can compute the execution time for several configurations in a row, just processing the same input trace.

We have extended this framework with the necessary changes to model multi-GPU load balancing and inter-GPU synchronization costs, and we have called it EMPATHY (standing for E + Multi-GPU Performance Analysis Tool + HY), shown in Figure 2 (right). The API state information from the input trace file related to surface management is used by EMPATHY to track whether RTT dependencies exist between frames or within the same frame. To model the screen split of SFR load balancing, pixel-related counters are simply scaled down by the number of GPUs as input to the proprietary analysis tool. For example, with 2 GPUs running in SFR we would cut down by half the number of Z tested fragments, since overall twice the Z test units are available working in parallel to consume the whole fragment workload. Since geometry-related counters remain unchanged, the SFR geometry scaling limitation is effectively simulated. Naturally, with this pixel downscaling, we model a perfect load balancing. We assume that current real SFR solutions may overcome the load-balance problem by using either screen splitting or tile-based rendering techniques, and get very close to our theoretical model. Required surface synchronizations for data dependencies in SFR are modeled and the penalization cost is added afterwards to get the total SFR execution time.

We model AFR load balancing using a different approach: assuming a perfect AFR load balance (consecutive frames taking roughly the same processing time), single GPU total execution time is divided by the number of GPUs and then added the penalization cost of inter-GPU synchronizations required in AFR. Combined AFR-SFR modes will first apply the SFR model on GPUs rendering the same frame and then the AFR scaling using the GPU clusters, following the data flow shown in Figure 2 (right). Both AFR and SFR performance models are fully explained in Section 5

The inter-GPU synchronization cost greatly depends on the bandwidth and topology of the GPU interconnection bus, and the transmission algorithm. The bus bandwidth is input architectural parameter to EMPATHY, set to the standard 6GBytes/sec PCIe 2.0 bandwidth for our experiments. Section 3.2 refers to a couple of usual GPU interconnection schemes and its related transmission algorithm, used in SFR mode.

2.2 The game workload

We have analyzed a representative set of popular DX9 games. The DirectX graphics API is widely used by 3D game engines, and DX9 drivers have supported multi-GPU for some time and thus are greatly optimized for these rendering modes. The selected games make diverse use of render-to-texture for environment reflections or shadow map effects. Some of them have been used in multi-GPU benchmark comparisons on websites [Cross 2006].

Table 1 summarizes the set of chosen DX9 games. The three last columns characterize the RTT surface usage: the total number of RTT surfaces used by the timedemo, the single GPU estimated total execution cycles in our proprietary tool, and the percentage of these cycles spent to render the RTT surfaces. For a proper evaluation, we used, for each game, the minimum screen resolution that guarantees that real execution is clearly GPU bound, which is required for multi-GPU scaling.

Game/Timedemo	Code	Engine/Developer	Release date	Screen resolution	Frames	RTT surfaces	Estimated cycles	% RTT time
3DMark 06/Canyon Flight	3DM06	Proprietary/FutureMark	2006/01	1600x1200x1AA	3344	23	49333M	69.11%
FEAR/Performance Test	FEAR	LithTech/Monolith	2005/10	1600x1200x1AA	4130	5	24923M	4.48%
Call Of Duty 2/carentan	COD2c	Proprietary/Infinity Ward	2005/10	1600x1200x1AA	1355	8	6409M	23.47%
Call Of Duty 2/demo5	COD2d	Proprietary/Infinity Ward	2005/10	1600x1200x1AA	1210	8	6591M	18.87%
Company Of Heroes/Intro	COH	Essence/Relic	2006/09	1600x1200x1AA	12195	5	62121M	85.74%
Half Life 2 Lost Coast/VST	HL2	Source/Valve	2005/10	2560x1600x8AA	9712	5	79281M	15.88%
BattleField 2142/suez canal	BF2142	Refractor 2/Refraction	2006/10	2560x1600x1AA	7400	8	38647M	80.05%
BattleField 2/abl-chini	BF2	Refractor 2/Refraction	2005/06	1600x1200x1AA	8217	3	75743M	4.15%

Table 1: *Game workload description.*

3 3D Rendering on Multi-GPU

3.1 Rendering load balance

The main goal of multi-GPU systems and the rendering balance modes is to evenly distribute the rendering workload among the available graphics processors. As in multi-core or multiprocessors systems, a good distribution should minimize the communication between GPUs and at the same time, keep all GPUs as busy as possible rendering the scene.

Hence, both the load balancing mode managed by the driver and the render-to-texture inter and intra-frame dependencies created by the graphics engine will determine the number of inter-GPU communications. Both classic multi-GPU rendering modes we present in the following subsections differ in the workload division and have different synchronization requirements.

Split Frame Rendering

Each available GPU is designated to work on a disjoint portion of the screen. Since the processing cost of each portion can vary significantly, the distribution must be done in such a way that no GPU is idle waiting for others to finish.

For 2 GPUs, a well-known solution consists in dividing the screen in two halves divided by a horizontal line that dynamically changes the Y-axis position to balance the rendering workload between the top and bottom half [NVIDIA 2009b]. This technique overcomes typical unbalanced scenarios such as landscapes with less rendering work above the skyline. The line Y-position is determined each frame based on the history of the previous frames, taking advantage of the fact that consecutive frames usually have only slight viewpoint changes.

In contrast, static assignment techniques try to avoid the time spent to analyze the history of previous frames. Tile-based load balance techniques divide the screen following a checkerboard pattern of very small tiles (e.g. 32x32 pixels). In general, partitioning of the screen in very small tiles ensures a good load balance of objects' pixel work, as shown in the SGI Infinite Reality[Montrym et al. 1997], which uses 2-pixel wide stripes to divide the image-space.

Due to render-to-texture dependencies, whether within a single frame or across multiple frames, all SFR techniques need the corresponding synchronization in order to provide each GPU with the full contents of the texture, because, in general, textures are not mapped and aligned to screen space; the sampled area of the texture does not necessarily correspond to the area rendered by the same GPU. In fact, the opposite situation is found more frequently: in the example of an environment reflection on the sea inverting a landscape image, the GPU in charge of rendering the sea in the bottom half of the screen will require the sky and mountain landscape rendered in the top portion by the other GPU. Some minor exceptions are for example the frame post processing filter passes, in which rendered and sampled regions are screen-aligned and perfectly split among GPUs. Because of the complexity of detecting filter passes just based on our gathered data, the analysis made in this work presumes that all RTT requires synchronization.

In the same way that we cannot determine a priori the sampled area of the texture, neither can we size up in an optimized way which subset of the geometry will be required within a tile or screen portion, since the projection of a primitive in screen space is computed as part of the rendering process in the GPU. It would require clipping at the object level of detail or even at the primitive level. So in practice, all the frame's geometry is always sent to all GPUs. That limits the scalability of the number of GPUs rendering in SFR, because given a fixed screen size, the more we increase the number of GPUs the less pixel workload each GPU has, and consequently the geometry part becomes the bottleneck in the rendering pipeline. Figure 3 shows the maximum scalability in SFR of our workload due to the limited geometry scalability. These scaling speedups were taken from EMPATHY before adding the penalization cycles for inter-GPU synchronizations (see Figure 2). Fortunately for multi-GPU scaling, most of our selected games (with exception of 3DM06 and BF2142) have a low geometry workload, helping the maximum SFR scalability of those games come close to perfect scaling.

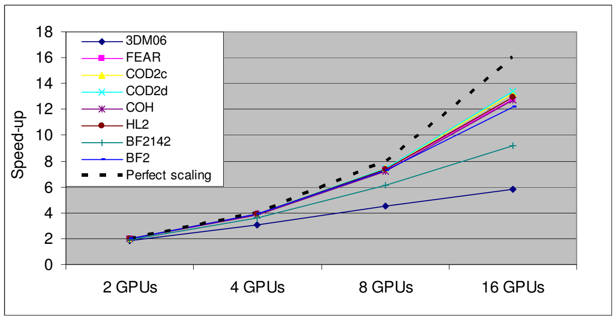


Figure 3: *SFR scalability limited by geometry.*

Alternate Frame Rendering

In this mode, each available GPU renders in parallel interleaved frames assigned to GPUs, usually in a round-robin fashion. Good load balancing is guaranteed as long as consecutive frames have similar workload, normally expected if there are no abrupt scene changes.

AFR has proven to be a very good load balancing method for self-contained frames, where all rendered surfaces are exclusively used in the same frame, and the contents are not persistent for the next frame. This feature of some game engines perfectly scales in AFR because each frame can be rendered independently; no surface dependencies exist between different frames, and thus synchronizations are not required. We can also avoid surface synchronizations as long as the dependencies exists within the same GPU, e.g. for two GPUs between only odd or even frames. In summary, dependencies eventually require synchronizations only if they cross GPUs.

Moreover, AFR rendering does not suffer from the geometry processing scalability problem of SFR. Each GPU only needs the list of primitives to render the corresponding assigned frames, and in this way, the geometry workload is perfectly split among the available GPUs. In summary, AFR is proven to be the best solution for game engines with independent frames or with surface dependencies skipping frames at the same distance than the GPU interleaving. Regarding that, in the results section we will see how since our games show almost no inter-frame dependencies, they scale perfectly in our AFR performance model.

However, AFR has another practical upper-bound limit in the number of GPUs rendering frames in parallel. The more available GPUs the faster the frame throughput, but the time to generate a single frame does not change. The noticeable effect is that the response time of the graphics engine to the user inputs is several times slower than the frame rate. Experiments have shown that, depending on the execution conditions, this input lag becomes annoying to the user in general with more than four GPUs.

3.2 The interconnection network

Figure 4 shows, for two and four GPUs, the arrangement of multi-GPU systems as a network of graphics processors among which the rendering task is distributed. The simple multi-GPU configuration is based on single-GPU graphics boards attached to the available PCIe slots on the motherboard 4(a). This way, each GPU is directly attached to the NorthBridge with an exclusive PCIe link, usually x16 lanes wide, giving a total bandwidth of 6 Gbytes/sec per direction for the new PCIe 2.0 [PCISIG 2007] for any communication between GPUs, CPUs and memory. To easily scale in the number of graphics processors, besides enabling more NB ports, instances of GPU pairs can be connected by a PCIe bridge within a graphics card, forming a hierarchical network (4-GPU example is shown in 4(b)). In either case, a display link daisy chain between GPUs allows the final composition of the display image from each GPU rendered framebuffer as the screen is refreshed.

GPU's local memories are better suited to store framebuffer data, and read-only data as regular textures and static geometry data. The main memory is the preferred location for dynamic data that are frequently changed by the CPU. Framebuffer surfaces are not shared provided that each GPU renders its own private screen-space portion, and having them in local memory greatly improves rendering performance (local GDDR memory bandwidth is higher than PCIe links's). Similarly, the RTT surfaces are allocated in local GPU memory to improve rendering, but these need to be synchronized through the interconnection network to every other GPU before the surface is read.

In order to synchronize the RTT contents, the writing GPU initiates peer-to-peer transfers to the required GPUs, in which the transmission packets are routed directly between the source and destination GPUs. In SFR mode, that involves all-to-all transfers of each rendered RTT screen portion to every other GPU, hence the number of transfers becomes $O(N^2)$ wrt the number of GPUs. For 2 GPUs though, only 2 peer-to-peer transfers are required, which can use the full peak bandwidth of PCIe links to the NB. In a 4-GPU system, each GPU initiates 3 transfers, a total of 12, to synchronize the 4 GPUs. Four of these transfers, actually occur between neighbors GPUs attached to the same PCIe Bridge skipping the NB and, if we assume that PCIe bridges can broadcast transfers to GPU pairs, then only half of the 8 remaining transfers cross the NB. Each of these 4

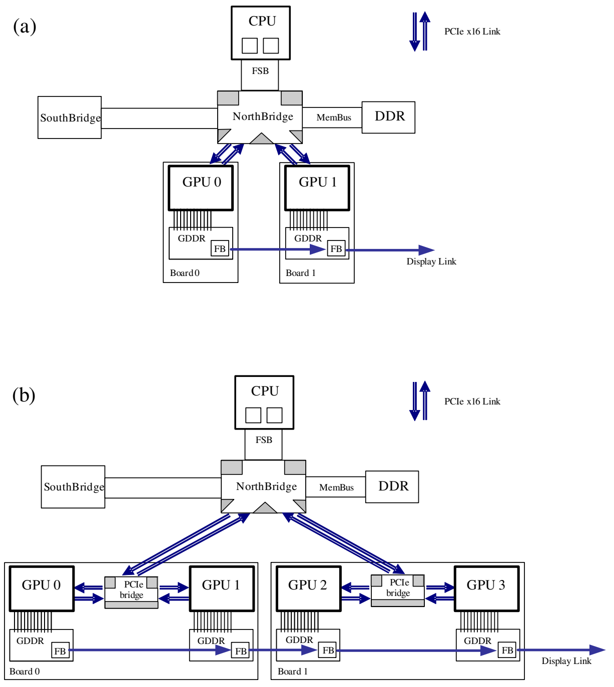


Figure 4: *a) 2-GPU and b) 4-GPU arrangement in dual-graphics card system.*

transfers (2 per direction) "sees" half of the peak bandwidth since the NB supports a maximum of two bidirectional transfers at the peak bandwidth. The following expression (used by EMPATHY to model synchronization costs in SFR), generalizes the previous concept of dual-GPU graphics boards attached to NB ports, with the NB becoming the bottleneck and reduced peak bandwidth based on the number of GPUs (≥ 4):

$$BW_{eff} = BW_{peak} * \left(\frac{N*(N-1)-N}{2}\right)^{-1}$$

Hence, whenever a synchronization is required, EMPATHY computes its transmission cost using the previous effective bandwidth and the corresponding bytes of the surface portion (Surface Bytes/N assuming perfect SFR split or tile-based SFR). For the present work, no other upload dynamic data or external traffic crossing the NB has been modeled, so EMPATHY considers the multi-GPU system to have exclusive use of the interconnection network.

3.3 Render Target Texture (RTT)

Target surfaces used in render-to-texture operations are called Render Target Texture surfaces in the D3D world. The graphics API provides with these 2D image resources to reuse rendering information in multipass algorithms such as environment reflection, shadow maps or GPGPU computation [Owens et al. 2008]. RTT surfaces can be written as a result of a rendering pass and read later as texture images, but never read and written simultaneously in the same render pass.

Sometimes the API programmer uses an offline plain surface (the usual surface used as backbuffer) as the render target, and the rendered contents are copied to a texture surface with different dimensions, which is eventually used for texture sampling. A special surface copy API call (StretchRect() in DX9) is used for this purpose.

A local shrinking/stretching filtering operation can be then applied to change the dimensions and/or the aspect ratio of the source. In this case, the surface that must be synchronized is the destination surface, also a RTT surface according to the DX9 specification [Microsoft 2002]. Our analysis framework also tracks StretchRect calls to deal with the dependencies created by the post-filtered surfaces in the same way as the usual RTT surfaces.

In this paper we do not analyze dependencies regarding any other kind of data updated in the rendering process. In dynamic data buffers, geometry is not preloaded in the GPU local memory but generated each frame by the graphics engine, as for particle systems and for vertex stream out, where new geometry is generated and stored inside the graphics pipeline. For the first case, dynamic data buffer use affects the traffic between the system and the GPU, but in just the same way as in single GPU rendering; for the second, vertex streaming out is not taken into account in this work since it is a new feature supported only in DX10.

In order to synchronize the RTT surfaces, the classic approach of graphics drivers with multi GPU support is to program a DMA transfer through the system PCIe interface to transfer the missing contents on demand; the action will be initiated no sooner than when the RTT surface is needed as input of a texture sampler. However, other alternatives can be explored as well.

In this paper we want to evaluate the potential benefits of anticipating the transmission of the surface contents with two alternatives. The first one programs the transfer right after the last render. Determining the last surface render is not trivial and requires some intelligence in the driver, for example, looking ahead in the command buffer, or using a prediction table based on history. Either way, it is not the intention of this paper to point to any concrete solution–we just want to evaluate the benefits of this capability. The point of this solution is that we do not need to pay the cost of the synchronization as long as the surface is not used immediately after the last render command and as long as the interconnection bus can support the transmission during this render-to-use gap. This gap can be thousands cycles if several intervening draw commands are not using the texture, or even millions of cycles if the next use is several frames later, far enough to completely hide the surface copy cycles. This is a feasible solution in GPUs having a DMA engine that can perform surface copies in parallel with the rendering pipeline.

3.4 Concurrent surface update

The second alternative to copy on demand comprises directing all surface pixel updates to the other GPUs through the interconnection bus at the same time they are updated in the local surface. Every GPU will listen to the bus and update its local surface copy accordingly. A potential drawback of this solution is that for surfaces with high overdraw, this technique will generate a lot of traffic.

However, if the interconnection bus can absorb the total required bandwidth, without increasing the execution time, then the solution is worthwhile. We expect that in many cases the cost of pixel shading, including the multiple texture samplings per pixel, will bound the rendering cost so that the cost of transmitting pixel updates will be partially or completely hidden. As a practical example, lets consider a pixel shader with 35 instructions (composed of both ALU and texture instructions) and one output instruction to write the final pixel color of 32 bits. Considering the modeled R600 architecture: 4 SPU clusters with 16 shader units each, the total shader throughput is a single SIMD instruction (vec4 + scalar) over 64 pixels each cycle (or equally assume a rate of 64 instructions per cycle over a single pixel), and also sampling texture data at very low cost (one cycle per access if hits in texture cache and using low cost bilinear filtering), the estimated cost for pixel processing would be:

Pixel shading cost:

$$35 instrs/pixel * \frac{1cycle}{64instrs} = 0.54cycles/pixel$$

Remote write cost:

$$4output.bytes/pixel * \frac{750Mcycles/sec}{6GB/sec} = 0.50cycles/pixel$$

Hence given a number of pixels, the cost of shading pixels at the R600 clock (750MHz) will be higher than the remote update cost and will hide the corresponding transmission cycles over the PCIe 2.0 bus (6GB/sec). Moreover, in real scenarios, texture accesses are not ideal due to expensive sampling filters that must fetch and operate on more than 4 texels per pixel. This supports even more our belief that the concurrent update cost can be hidden in most of cases by the pixel processing cost, especially for those game workloads using long pixel shader programs or very expensive texture accesses. Later, in Section 6, we will confirm the success of this approach combined with early copy of surfaces, running on real game workload's shaders and even larger than 32 bit output pixel formats.

A third alternative is worth mentioning for future investigation. Since virtualization of the memory address space is becoming more important for new driver models that transparently manage the available graphics resources [ATI/AMD 2006], it will be possible to read RTT surfaces directly from other GPUs, taking advantage of a single shared memory space with an underlying distributed memory system. The biggest challenge with this alternative is to deal with the tremendous variance in texture read latency of pixel shaders. In [Moerschell and Owens 2006], the authors propose for texture data, such a distributed virtual memory system with page invalidation and transparent data migration to improve the access to reused texture pages. Today both AFR and SFR approaches assume separate address spaces per GPU, and several address translations need to be performed in order to reach the destination address.

4 RTT Characterization

4.1 Surface rendering and use

To analyze inter and intra-frame dependencies of RTT surfaces, we first look at the related API state calls. Table 2 summarizes the API calls gathered for BF2142's surfaces (identified with a set of internal numbers consecutively assigned by EMPATHY). Columns 1, 2 and 5 show respectively SetRenderTarget, SetTexture and Clear calls on the respective surfaces. Columns 3 and 4 show the counts for the involved source and destination surfaces of a StretchRect operation.

SetRenderTarget assigns the surface for writing ("Render"), SetTexture assigns it for reading ("Use"), and Clear initializes its contents to a default value. In case of the first two, since they only change an API state, they don't indicate actual writing and reading (only the final state really matters); thus the actual writes and reads are shown in the 'Real Usage' columns. The occurrence of more API calls than real uses shows an inefficiency due to redundant state calls that could be avoided.

As we can see, in general some counts are closely related to the number of frames; for instance, comparing to the number of frames in Table 1, surfaces 308 and 309 are written and read roughly twice a frame. The first surface in the list is marked with an asterisk, meaning that this is not a RTT surface but a regular off-screen surface. Its contents turn out to be copied, using a StretchRect call, mainly to RTT surface 307, which is rendered directly and read many times too.

	API					Real Usage		Dependencies					
Surface Id	SetRenderTarget count	SetTexture count	StretchRect (dest)count	StretchRect (src)count	Clear count	Write count	Read count	RtU distance = 0 count	syncs	Avg sync gap	RtU distance $>$ 0 count	syncs i2	syncs i4
305*	14823	0	0	14811	0	67654	0	0	0	14.60%	0	0	0
306	7406	25155	1	0	7405	297054	335442	335442	7405	18.00%	0	0	0
307	14812	27125	14810	0	7405	1446827	35722	35722	28322	0.00%	0	0	0
308	14811	14805	0	0	1	14810	14805	14805	14805	0.00%	0	0	0
309	14812	14811	0	0	1	14811	14811	14811	14811	0.00%	0	0	0
30a	2	2	0	0	1	1	2	2	1	100.00%	0	0	0
30b	2	1	0	0	1	1	1	1	1	100.00%	0	0	0
30d	7306	12009	0	0	7306	126290	18523	14516	4713	18.70%	4007	9	26
30e	7306	39004	0	0	7306	118530	252642	252642	7305	2.70%	0	0	0

Table 2: *Surface related API calls and real usage for BF2142.*

Under the 'Dependencies' tag of Table 2, 'count' columns show how many Render-To-Use dependencies exist, broken down by dependencies in the same frame (RtU distance = 0) and dependencies where the Render and the Use happen in different frames (RtU distance > 0). Each read creates a dependency with the last of a sequence of consecutive writes, so the total dependencies match exactly with the number of reads. Intra-frame dependencies impact SFR performance while the inter-frame dependencies will determine AFR scaling.

Render-To-Use gap within the same frame

Render-to-Use dependencies within the same frame occur between the last render on a surface to every subsequent use. However, since all the consecutive uses within the frame are made by the same group of GPUs, only the first use actually requires a synchronization to update the surface's contents. 'Syncs' columns in Table 2 complete the dependency information with the corresponding synchronizations. Another very important measure for performance evaluation is the time gap between the render and the use of synchronizations. The larger the gap, the more time there is to synchronize without penalty. The third column under the 'Dependencies' tag of Table 2 shows, per surface, the average percentage of frame time that synchronization gaps represent. Considering the extreme cases, surfaces rendered at the beginning of the frame and used at the very end will have 100% gaps, while surfaces used immediately after the draw command will have 0% gaps. If multiple synchronizations exist in the same frame for the surface, the cycles for multiple gaps are accumulated.

Render-To-Use frame distance histogram

In contrast, when Render-to-Use dependencies are between different frames, different GPUs may be in charge of the source and the destination frames. If it is the case, as explained before, the first use will require synchronization, and subsequent uses will not, as long as they also happen in the same destination frame. In addition, if the same GPU is in charge of rendering both source and destination frames, synchronization is not required. The rightmost columns in Table 2 compare the Render-to-Use dependencies at a distance greater than 0 with the actual required synchronizations when frames are assigned to GPUs interleaving by 2 frames and by 4 (AFR rendering). As we can see for the surface 30d, interleaving by 4 requires more synchronizations than interleaving by 2, because in the latter, each GPU renders at twice the frequency, so the probability of the same GPU rendering the source and destination frames of the dependency is greater.

Summarizing inter-frame data in Table 2, the number of required inter-frame synchronizations is very low; thus, in such cases AFR performance will scale perfectly. Figure 5 breaks down, for the BF2142's 30d example surface, the synchronizations for interleaving by 2 and by 4, according to the frame distances.

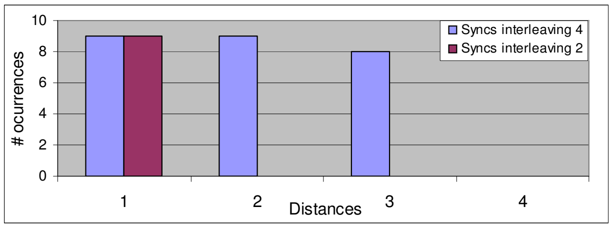


Figure 5: *Sync distance histogram for BF2142 surface 30d.*

As we can see, all the synchronization distances turn out to be very close. This graph, in conjunction with the data in Table 2, shows us that, for the great majority of the dependencies at a distance greater than 0, the writer and reader GPU are the same, and for the remaining that cross GPU boundaries, writer and reader frames are really close (3 frames at most). Note that all the synchronizations (9) at distance 2 are avoided in interleaving by 2 since each GPU renders every two frames and also those at distance 3, because the GPU that renders frames at distance 1 and distance 3 are the same; hence only the first use requires a synchronization.

4.2 Synchronization cost

In this section we evaluate the best synchronization alternative for each surface based on the previously explained characterization. Since our characterization is per surface and at the frame level of detail, we take the best single synchronization decision for a surface every frame. This decision is based on the flowchart in Figure 7 and determines the synchronization cost. However, to understand the overhead computations of this decision diagram we must first take a look to the graphs in Figure 6 for the example of BF2.

The first graph compares the accumulated copy transmission cycles of all the intra-frame synchronizations to the total synchronization gap cycles of a given surface. The copy transmission cycles C are computed as follows:

$$C = I + \frac{SB}{BW_{eff}} \tag{1}$$

where I is the initial copy setup overhead (in cycles), BW_{eff} is the effective interconnection bus bandwidth, as computed in Section 3.2 and SB the total surface bytes (width x height x pixel depth). The penalization cost of using the early copy alternative is the result of subtracting the sync gap cycles from the copy cycles and clamping to 0 as the lower bound, so if we have enough gap we do not pay

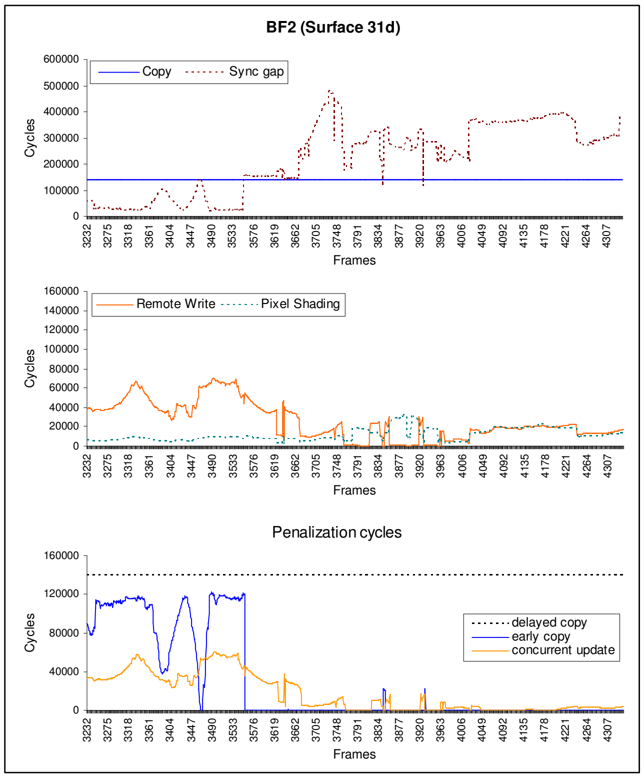
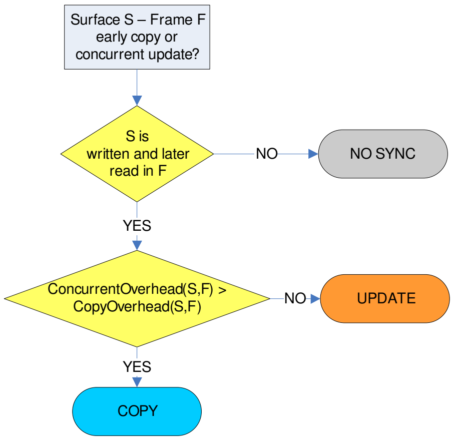


Figure 6: *Synchronization graphs for BF2 surface 31d.*


Surface S – Frame F early copy or concurrent update?
S is written and later read in F
NO
NO SYNC
YES
ConcurrentOverhead(S,F) > CopyOverhead(S,F)
NO
UPDATE
YES
COPY

$$ConcurrentOverhead(S,F) = \max(0, RemoteWrite[S,F] - PixelShading[S,F])$$

$$CopyOverhead(S,F) = \max(0, Copy[S,F] - SyncGap[S,F])$$

Figure 7: *Early Copy vs Concurrent decision flowchart.*

the copy cost. The second graph compares, for the same surface, the accumulated cost of remote writes to the accumulated pixel shading cycles. In this case, the penalization of the concurrent update alternative is computed by subtracting the pixel shading cost from the cost of remote writes, according to that exposed in Section 3.4. So clamping again to 0 for the lower bound, if pixel shading cost is high enough, remote updates will not result in penalization. Remote updates cycles are computed using the same interconnection bandwidth in Equation 1. The third graph compares penalization cycles of both alternatives in addition to the copy transmission cycles C, considered directly the penalization cost for delayed copy. As we can see in the example case, both proposed alternatives greatly reduce the synchronization overhead and choosing the best alternative each frame results in the minimum penalization.

5 Multi-GPU Performance Model

5.1 SFR and AFR penalization models

In the previous section we disclosed the characterization on which EMPATHY's SFR and AFR synchronization penalties are based. The SFR model adds the accumulated penalization cycles of all the RTT surfaces throughout the execution, to the runtime cycles after downscaling pixel counters. The flowchart of Figure 7 is used to choose the synchronization technique that results in the minimum penalization each frame. The SFR model uses Equation 1 to compute the transmission cost with the total surface bytes SB divided by the number of GPUs N, since this is the rendered surface portion that each GPU is sending to every other GPU. The following expression summarizes SFR execution time (in cycles):

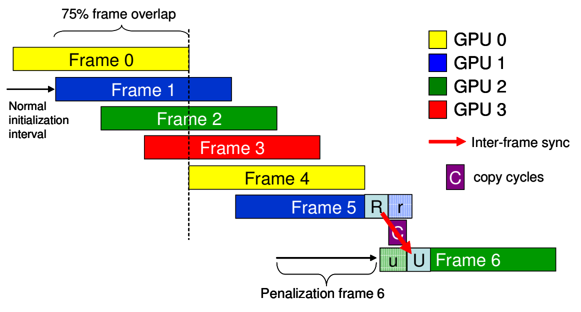
$$SFR_{cycles}(N) = PxScaledGPU_{cycles}(N) + \sum_{\forall surface S} \sum_{\forall frame f} Penal_{SFR}(N,S,f) \quad (2)$$

Good SFR scaling is achieved by having good initial geometry scaling ($PxScaledGPU_{cycles}(N)$ gets close to $SingleGPU_{cycles}/N$) and by reducing the penalization cycles, for example by using high interconnection bandwidth, or when the workload has spaced-out synchronization gaps and/or it is highly pixel shading bound.

The AFR model presented in this paper uses inter-frame dependency information to compute the actual gap for each required synchronization, taking into account overlapped execution of frames. Specifically, as long as consecutive frames are assigned to different GPUs working concurrently, synchronizations that in serialized execution go forward in time, in concurrent execution can even go backward in time. The extreme example of the latter case is when a surface is written at the end of the frame and read at the very beginning of the next one. In order to ensure that a GPU does not read a surface before the contents are rendered and sent through the interconnection bus, the GPU must wait (synchronize) for the other GPU's writing to complete, as depicted in Figure 8 for the 4 GPU AFR case. These wait cycles are the penalization cycles that impact on AFR performance.

Only synchronizations within overlapping frames have been taken into account in our model: as shown in Figure 8, starting from frame 0, only 3 frames actually overlap (interleaving - 1) and neither frame 4 that is rendered again by GPU 0, nor the following non-overlapping frames that are too far away to cause a penalization, are considered. For the latter case we assume that if the memory and interconnection bus are not fully utilized, then the copy operation can be performed in a reasonable time within this interval without penalization.

In case of synchronizations between overlapping frames, the bottom of Figure 8 shows the recursive expression to compute the penalization cycles of every frame. The first part of the aggregate con-



$$Penal(N,j) = \max_{\forall syncs(i,j)} (overlap(N,i,j) * cycles[j] + C - (r + u + \sum_{\forall k, i<k<j} Penal(N,k)))$$

Figure 8: *AFR penalization model.*

sists in the penalization in the worst case (render at the end and use at the beginning), computed as the cycles of the destination frame that overlap the source frame. The $overlap(N,i,j)$ auxiliary function computes this overlapping percentage between frames i and j for an interleaving of N GPUs, and the $cycles$ array stores the frame execution cycles of the serialized execution. Copy cycles also contribute to the penalization and are computed using Equation 1 and the whole surface area. The second part alleviates the penalization by subtracting any existing cycles from the render to the end of the source frame (the portion called r) and subtracting the gap from the beginning of the destination frame to the point where the surface is actually used (the portion called u). In addition, if penalizations have been computed for previous frames between the source and destination frames, then these cycles are deducted as long as penalizations delay the rendering of the waiting and also subsequent frames. The expression is recursively called to determine that previous penalizations. Finally, given a frame, the maximum penalization for all the synchronizations involving any surface will count as the frame's AFR penalization.

In order to compute the total penalization, the AFR model computes progressively and recursively the penalization expression across frames, and eventually, the total AFR execution time is computed as follows:

$$AFR_{cycles}(N) = \frac{SingleGPU_{cycles}}{N} + \sum_{\forall frame f} Penal_{AFR}(N,f) \qquad (3)$$

This time, as shown in Expression 3, the single GPU execution cycles are perfectly scaled down by the number of GPUs since the geometry processing is also split in AFR. And the key for small penalizations is to have few inter-frame synchronizations. Early in the AFR analysis we stopped experimenting with the synchronization alternatives we proposed for SFR because in our AFR model, the copy cycles have shown to have little effect in the penalization compared with the effect of r and u, i.e. the render and use positions within the corresponding frames. In addition, most of the workloads turned out to have very few inter-frame synchronizations. COH and HL2 are the exceptions; however, the penalization percentages observed in these workloads are so negligible that the full experimentation with the alternatives would not reach a satisfactory conclusion.

5.2 Combined AFR-SFR evaluation.

The expression for combined AFR-SFR modes simply takes the corresponding number of GPUs that forms a cluster and computes the SFR part using Expression 2. The resulting execution cycles are then used as single GPU execution cycles input in Expression 3 in order to compute the final combined scaled execution with the remainder GPU clusters. A functional advantage of this approach is that it allows EMPATHY to reuse computations when evaluating several configurations together (the evaluation of the 2 SFR - 2 AFR (4 GPUs) combined mode can reuse part of a 2 SFR pure mode evaluation for the same workload). The following consolidates both previous expressions into a combined SFR-AFR runtime expression, that formally describes the EMPATHY's calculation flow shown in Figure 2:

$$SFR.AFR_{cycles}(N,I) = \frac{SFR_{cycles}(\frac{N}{I})}{I} + \sum_{\forall frame f} Penal_{AFR}(I,f)$$

where I is the AFR interleaving or number of GPU clusters, and correspondingly $\frac{N}{I}$ is the number of GPUs that render a single frame in SFR within a cluster.

6 Experimentation Results

Figure 9 summarizes the scaling results with different numbers of GPUs and rendering mode configurations, using the early + concurrent synchronization technique presented in this paper. The left side shows those pure AFR and combined SFR-AFR configurations in which the number of GPUs assigned to AFR is greater than or equal to the number assigned to SFR. The right side shows those configurations with the majority of GPUs rendering in SFR mode (including pure SFR). Note that AFR scalability has been limited to four GPUs in order to skip configurations suffering from excessive latency. The Y-axis represents for each configuration the average percentage scaling for all workloads, relative to perfect scaling for that configuration.

In general, the left side configurations (mostly AFR) beat the right side's (mostly SFR), especially pure AFR configurations, which achieve perfect scaling. This is not surprising since our selected workload turned out to be well-suited for AFR modes because most of the RTT surface dependencies and related synchronizations happen within the same frame, as summarized in Table 3; so the one-to-one assignment of GPUs to frames in AFR clearly benefits performance. The non-pure AFR configurations have a performance drop that bring them closer to the "mostly SFR" configurations, although, any "mostly AFR" configuration is always faster than the corresponding configurations with the same number of GPUs on the right side. Following the same reasoning, SFR pure configurations have the worst performance and improve as we incrementally assign more GPU clusters for AFR.

The second goal of this paper–the evaluation of early copy and concurrent update techniques versus delayed copy of RTT surfaces–has been summarized in Figure 10. It shows the average relative speed-up of the early copy technique used separately and the early copy combined with concurrent update, normalized to the delayed copy performance. Average of 4 SFR, 8 SFR and 16 SFR configurations has been taken since the concrete experiments with different number of GPUs have shown proportional speed-up with similar performance differences among techniques.

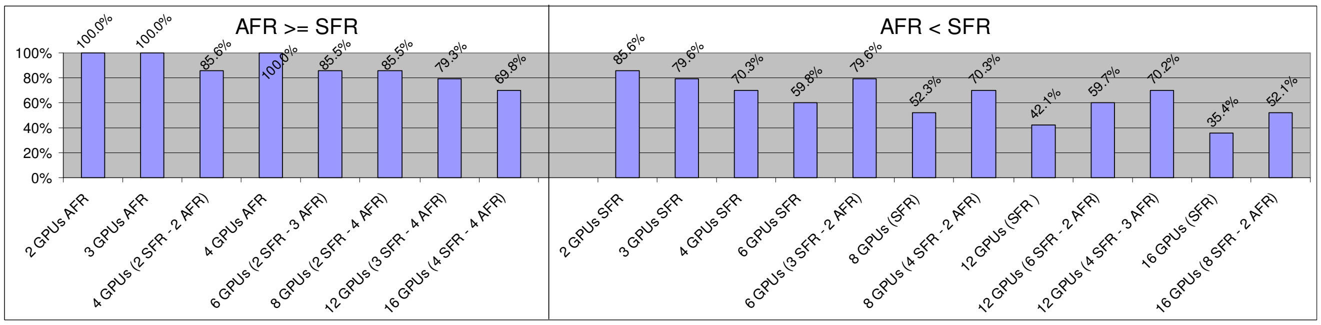


Figure 9: *Average efficiency of SFR, AFR and combined SFR-AFR modes.*

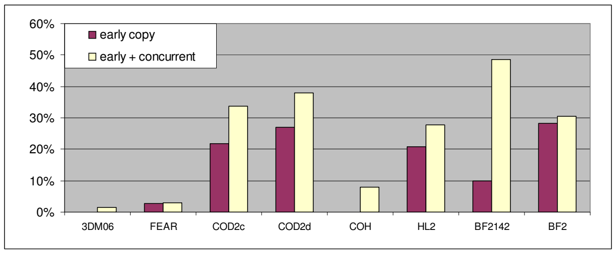


Figure 10: *Average speed-up gain of the proposed synchronization alternatives.*

	Intra-frame			Inter-frame	
Game/ Timedemo	syncs	weighted sync gap	weighted pixel shading	syncs i2	syncs (i4)
3DM06	123691	0.42%	52.85%	0	0
FEAR	2426	83.68%	65.08%	0	0
COD2c	83780	1.70%	96.28%	0	0
COD2d	48175	1.65%	94.63%	0	0
COH	36626	0.01%	96.80%	41	83
HL2	19788	36.45%	21.95%	1442	1945
BF2142	77363	3.13%	87.16%	9	26
BF2	22553	30.40%	44.44%	1	3

Table 3: *Game synchronization summary.*

The success of combined early copy and concurrent update can be explained with the data in Table 3. This table shows in the second column the total number of intra-frame synchronizations for all the workloads, and in the third and fourth columns the weighted synchronization gap and weighted pixel shading percentages. The weighted synchronization gap is the average of the same per-surface measure seen in Table 2, weighted according to the number of inter-frame synchronizations, so less synchronized surfaces have little effect on the average. The weighted pixel shading shows in average, which percentage of the cycles spent to render RTT surfaces was pixel shading bound, similarly to the analysis made in Figure 6. These two overall characterization measures correspond to the single GPU execution and they can be seen as good indicators of the application's suitability for each technique.

For instance, COH has no improvement for early copy because there is little sync gap, so ends up having the same penalization as delayed copy. In contrast, concurrent update performs better as long as it has a high weighted pixel shading percentage. However, it is not comparable to the outstanding improvement in BF2142 even having similar weighted percentages. The reason is that the number of synchronizations in COH is roughly half, so this lower weight of the penalization on the execution gives fewer opportunities to the concurrent update technique to improve the overall performance. In the extreme, FEAR has a negligible number of synchronizations, hence both techniques barely make any improvement.

In contrast, HL2 and BF2 show a relatively good improvement for early copy (better in the latter again due to the higher number of synchronizations) and almost no gain using early copy + concurrent update. Here, while 30% of the frame is considered a very spaced-out gap (most of the high resolution surfaces can be copied within these gaps), a weighted pixel shading below 50% does not help concurrent update to raise early copy's mark. Finally, both COD2 timedemos show relatively good speed-up in early copy even though average gaps are quite short. Coincidentally, in these cases, the surfaces involved in those minimal gaps are very small (128x128 pixels), so transmission cycles fit quite well in them.

7 Related Work

Little work has been done on the characterization and scalability of modern 3D applications. In [Sibai 2007], the author scales different system resources as number of CPUs, number of GPUs and number of threads supported by the hardware to analyze the scalability of a well-known benchmark for graphics subsystems, 3DMark. Due to the multi-threaded nature of this benchmark, especially due to its Physics and AI subsystems, the paper concludes that increasing multiple thread capacity of the systems results in a better scalability of this workload. The characterization made in this paper is limited to API call analysis and memory usage of a single workload and the GPU scaling is limited to a dual-graphics card system. Our work, however, analyzes surface synchronization requirements as the main determinant factor for scalability of a list of modern 3D games, and based on this characterization, evaluates forthcoming realistic multi-GPU configurations using more than 4 GPUs.

In contrast, a lot of previous work has been published regarding high-performance parallel rendering systems, both using clusters of multiple computers [Humphreys et al. 2002] and specific architectures with dedicated interconnection systems [Molnar et al. 1992][Eldridge et al. 2000][Montrym et al. 1997]. All of them face the same old problem of evenly distributing the workload across GPUs. The main difficulties in this task are: 1) evenly balancing the triangle and pixel work among units; the latter is a major challenge in graphics processing because the amount of work required for a set of primitives is not known a priori, 2) minimizing work replication in the different processing to improve scalability. Other related problems are specific to the graphics system implementation, such as the scalability of the pixel interconnection network to blend the final displayed image, especially on those image composition based systems [Molnar et al. 1992][Springer et al. 2007].

A taxonomy for parallel rendering systems [Molnar et al. 1994] classifies all these systems according to where the sorting of rendering workload is performed in the pipeline. Sort-first systems distribute the workload to the units responsible of the respective assigned screen region, at the early primitive stages. Examples are Chromium [Humphreys et al. 2002], the task adaptative dynamic scheduling approach presented in [Whitman 1993] and, in some way, NVIDIA's SLI [Relations 2005] and AMD's Crossfire [Persson 2005] commodity multi-GPU systems. Sort-middle systems transform primitives arbitrarily in any processing units and sorts them to the corresponding screen-region rasterizer, as SGI InfiniteReality [Montrym et al. 1997]. Sort-last systems is the natural option for image composition systems as PixelFlow [Molnar et al. 1992] or Multi-Frame Rate Rendering Techniques [Springer et al. 2007] in which final processed pixels are sorted to the blending processors responsible of each screen region. Other proposed architectures, as Pomegranate [Eldridge et al. 2000] combine sorts at different points of the pipeline to get a more scalable solution.

Each system alternative has different communication requirements and balancing issues, and several proposed solutions address all these problems. Most of them assume, however, a strict linearity not existent in today's process of rendering. For instance, as we have seen in this paper, current 3D applications reuse rendered data in multi-pass algorithms, introducing an important synchronization overhead. Furthermore, geometry binning to screen-space used in [Whitman 1993] becomes much harder with the introduction of programmable Vertex and Geometry Shading stages. Hence, a review of these parallel rendering systems is needed for their applicability in interactive rendering with state of the art 3D engine technology.

8 Conclusions

We have introduced a novel game characterization for multi-GPU execution, based on the inter-GPU synchronization requirements of Render Target Texture surfaces. The RTT characterization explained in this paper has been used to evaluate the performance of rendering load balance configurations for multi-GPUs, using a proprietary analytical tool. According to our analysis, the great majority of RTT synchronizations in our workloads happen within the same frame, and that benefits the performance of pure AFR modes due to the one-to-one assignment of GPUs to frames.

It was also of a special interest the evaluation of combined AFR-SFR rendering modes that overcome the practical limitation of pure rendering modes allowing the use of a larger number of GPUs. We have obtained above 50% of efficiency compared to the perfect scaling for this combined AFR-SFR modes, which improve up to 80% in average when the configurations are mostly AFR.

Finally, by introducing early copy and concurrent surface update techniques as alternative to the classic delayed surface copy, our analysis has shown how the former turns out to be very beneficial in SFR modes, especially for those workloads in which most of the time spent to render RTT is pixel shading bound, thus hiding the cost of remote updates.

Acknowledgements

This work has been supported by the Spanish Ministry of Education and Science under grant TIN2007-60625.

References

ATI/AMD, 2006. Graphics Drivers and WDDM: Leading the way to Windows Vista^{TM}. `http://ati.amd.com/products/wp/ATIWDDMWhitepaperFinalV38.pdf`.

CROSS, J., 2006. ExtremeTech SLI vs. Crossfire Benchmarks. `http://www.extremetech.com/article2/0,1697,2010874,00.asp`.

ELDRIDGE, M., IGEHY, H., AND HANRAHAN, P. 2000. Pomegranate: a fully scalable graphics architecture. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 443–454.

HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. 2002. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 693–702.

MICROSOFT, 2002. DirectX 9 SDK Specification.

MOERSCHELL, A., AND OWENS, J. D. 2006. Distributed texture memory in a multi-GPU environment. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, ACM, New York, NY, USA, 31–38.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. Pixelflow: high-speed rendering using image composition. *SIGGRAPH Comput. Graph. 26*, 2, 231–240.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl. 14*, 4, 23–32.

MONTRYM, J. S., BAUM, D. R., DIGNAM, D. L., AND MIGDAL, C. J. 1997. InfiniteReality: a real-time graphics system. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 293–302.

NVIDIA, 2009. NVIDIA PerfHUD - Developer Site Homepage. `http://developer.nvidia.com/object/nvperfhud_home.html`.

NVIDIA, 2009. NVIDIA SLI Zone FAQ. `http://www.slizone.com/page/slizone_faq.html`.

OWENS, J., HOUSTON, M., LUEBKE, D., GREEN, S., STONE, J., AND PHILLIPS, J. 2008. GPU Computing. *Proceedings of the IEEE 96*, 5 (May), 879–899.

PCISIG, 2007. PCI Express 2.0 Base Specification.

PERSSON, E., 2005. Programming for CrossFire^{TM}. `http://ati.amd.com/developer/SDK/AMD_SDK_Samples_May2007/Documentations/Programming_for_CrossFire.pdf`.

RELATIONS, N. D., 2005. NVIDIA GPU Programming Guide. `http://download.nvidia.com/developer/GPU_Programming_Guide/GPU_Programming_Guide.pdf`, July. 2.4.0 ed.

SIBAI, F. N. 2007. 3D Graphics Performance Scaling and Workload Decomposition and Analysis. *Computer and Information Science, ACIS International Conference on 0*, 604–609.

SPRINGER, J., BECK, S., WEISZIG, F., REINERS, D., AND FROEHLICH, B. 2007. Multi-Frame Rate Rendering and Display. *Virtual Reality Conference, 2007. VR '07. IEEE* (March), 195–202.

WHITMAN, S. 1993. A task adaptive parallel graphics renderer. In *PRS '93: Proceedings of the 1993 symposium on Parallel rendering*, ACM, New York, NY, USA, 27–34.