Yuanfang Xia, 21421277. This paper was done in 2013 in SJTU and submitted to ICCD 2014, later withdrawn.

CG-Sharing: Efficient Content Sharing in GPU-Based Cloud Gaming

Abstract

With the fast development of the GPU server technology, cloud gaming has come to the era and become popular in recent years. Unlike the traditional desktop gaming where the graphic rendering is performed locally using the user's personal graphics card, cloud gaming environment tends to run distributed games to support many users at the same time in the cloud gaming center where most of the rendering jobs are done in the remote GPU cluster. The rendered frames are streamed to user's devices such as notebooks, tablets and cell phones. For the economic cloud gaming to be viable, the operator must make full utilization of the expensive hardware resources like the graphic cards, and the state of art technology tries to render multiple instances of games on the same GPU. In this paper, we first identify that there are many redundant and duplicated contexts/workloads existing in today's cloud gaming rendering that waste a large amount of memory bandwidth and system energy. We in turn propose novel system and architecture enhancements to effectively share the contents across the game instances from different users in the cloud gaming center. Our experiment shows 31.84% decrease in the DMA transfers, 26.89% reduction of video memory usage, 8.99% increase in the gaming performance and 8.51% cutdown of the system power consumption in the GPU server using our proposed content sharing methodology.

1. Introduction

Cloud gaming [45], also known as Gaming on Demand or Gaming as a Service, is an online gaming style that allows on-demand streaming of games onto target devices such as notebooks, tablets and cell phones. This is similar to Video on Demand. The control events from mice, keyboards, joysticks and touch-screens are transmitted from the devices back to the cloud gaming servers. Unlike the traditional gaming experience where the graphic rendering is performed locally using the player's own graphic card, the actual scenes in cloud gaming are rendered in the GPU servers and are streamed directly to the client. The most commonly used technology for cloud gaming are video streaming and file streaming. For most of the popular cloud gaming platforms [33][16], video streaming is chosen because it is less demanding in terms of the device capability and bandwidth.

It is a rather new concept that recently gains popularity with consumers due to the ever-improving availability of sufficient networking infrastructure to meet the high requirements for bandwidth and latency. This service takes the advantage of the broadband connection, powerful server clusters and the video stream compression to stream the high-quality game contents to a subscriber's device. Users can play without acquiring the actual game because the game code is executed primarily in the server cluster. As a result, the subscriber can use a less powerful device to play a fancy 3D game which is normally not possible to run locally, since the server does all the heavy-lifting workloads for subscriber's device. Most cloud gaming platforms are private and proprietary such as OnLive and Gaikai [33][16]. The first open source cloud gaming platform was not released until April, 2013 [17]. The forecasted cloud gaming market value can be as high as 24.75 Billion USD [40].

On the other hand, GPU vendors such as NVIDIA have begun to provide cloud gaming data centers with custom hardware targeting at a greater efficiency for concurrent games per grid node. With NVIDIA Grid technology [31], the first product that supports cloud-gaming, the GPU utilization ratio is about two simultaneous users per graphics card. For each server rack, it can hold 480 users and consume between 800 and 900 watts power per blade [11]. Obviously, this kind of low hardware utilization ratio and power-hungry cloud gaming service is not financially competitive enough for ordinary users to switch from conventional gaming style to the more advanced cloud gaming [1][11].

One major reason behind that is today's cloud gaming products are architecturally based on commercial graphics cards which are designed for one application at a time. These products fail to recognize the fact of the concurrency between different users who play the same game in the cloud gaming environment. Due to the nature of the "single player/single device" model prior to the advent of cloud gaming, a system was rarely required to execute multiple instances of a single game concurrently. Existing resource management solutions from the hardware architecture to the system driver mostly consider the optimal usage of GPU resources for a single application, but fail to take into account that applications from multiple users might operate on a big set of overlapping data. For example, texture data are initially stored in the main memory of server, so every overlapped access of texture will force the GPU to fetch superfluous texture data from main memory to the GPU's local video memory. Current cloud gaming solution treats every instance of the same game as stand-alone that potentially causes large wastes of the system power and memory bandwidth.

Imagine a classic arena-type multi-player game such as "Doom 3", which is among one of the most popular games supported by the cloud gaming platforms. According to our observations, the typical procedure in the beginning of the gaming session is to copy all textures that belong to the are-

na and the involved player models from the system memory space into the GPU video memory space. This will be the same static data for each client participating the game, but there is currently no way for the GPU driver to know about that. This unawareness leads to duplicated memory usage and superfluous memory transfer operations. Also, when a new game instance is loaded into the GPU pipeline, it will "flush" the entire cache hierarchy including the texture cache wasting all the static texture data that may contribute to the next hit. Hence, it is imperative that we tweak the driver and modify the architecture to adapt the GPU design to the cloud gaming environment.

In this paper, we first performed a static analysis to show how many unexpected redundant data operations actually occur in a GPU server by intercepting and recording draw calls of several main-stream games through 3D API. This proves the large potential of the content sharing in a cloud gaming center. We further proposed corresponding driver changes and architectural modifications to recognize and minimize the redundant data operations which can save a significant amount of memory bandwidth, memory space and system power consumption.

To the best of our knowledge, this is the first work to analyze the commercial games for content sharing in the multi-user cloud gaming environment and propose system/architecture enhancements to optimize the efficiency in the GPU server cluster. In summary, our contribution is two-folded:

- We raise the issue of the heavily superfluous data operations in the current GPU-based cloud gaming solution and show how much they can affect the system efficiency.
- We propose a system technique to identify the redundant data operations and an effective way of changing the driver to share contexts among different user instances. We implement the total solution in a real-time cloud gaming system and demonstrate the benefit by real system measurement.

The remainder of this paper is organized as follows: Section 2 describes the background. In Section 3, we provide static analysis to show the potential of our work. In Section 4, we discuss our system and architectural solutions. Section 5 describes experiment setup. Section 6 shows our experiment results. In Section 7, we provide the related work. Finally, Section 8 draws the conclusion.

2. Background

This section provides background knowledge on GPU game rendering and the basic concepts for cloud gaming.

2.1. GPU Game Rendering

Conventional games are rendered on individual user's local GPU which typically allows a single game instance running at a time. GPU architectural features are exposed to the programmers through graphics APIs (OpenGL or Direct3D). The APIs provide the interface with all the required functions to render

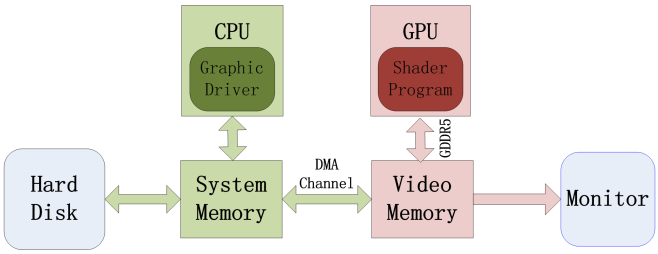


Figure 1: GPU game rendering procedure

a 3D scene to be displayed on the screen including allocations for memory objects like textures and triangle vertices.

In a gaming application, GPU rendering is usually processed in a loop. There are three stages in the loop. As shown in Fig. 1, the first stage is processed in CPU where the graphic driver is run on. The driver prepares the dataset for the later GPU rendering. This includes fetching textures from hard disk, calculating objectives primitives for the upcoming frames and preparing the shader programs. The prepared data is stored in the system main memory and then transferred to the GPU video memory through the DMA channels. The second stage is the major frame rendering stage where GPU does all the computational intensive works and updates the pixel values in the video memory according to the shader program. The final stage commits the commands from the GPU API calls and outputs the contents in the frame buffer to the monitor [38] [29] [35] [3].

Modern graphic pipelines are fully programmable. They execute shader programs which are often compiled into an intermediate bytecode represented by either the API library or a front-end part of the driver. The graphic driver is responsible for keeping track of the API states and translating API calls into native commands the GPU hardware understands. Furthermore, the driver requests space for object allocation from either its own privileged component or the operating system kernel and decides where and how those objects should reside in the system or the video memory [35] [3]. The commands and memory objects are written into specific locations in the video memory from where they get picked up by the graphic execution engine. The shader units in the engine can be programmed to render a 3D scene. Interrupts are used to notify the driver of events like a finished frame execution.

In the graphic system, the memory bandwidth can often be the limiting factor since modern 3D games require huge amount of memory transactions. For example, a GPU requires very high memory throughput with nearly 180 GB/s memory bandwidth on a GeForce Fermi architecture [32] while an *Intel Nehalem* platform operates on a maximum of 32 GB/s [23]. As shown in Fig. 1, there are two places that can be bandwidth intensive. The first one is the DMA channel between the system memory and video memory limited by the maximum bandwidth of PCIE buses. The second one is the memory buses between the video memory and the GPU core. The limitation here is the bandwidth of GDDR5 data rate.

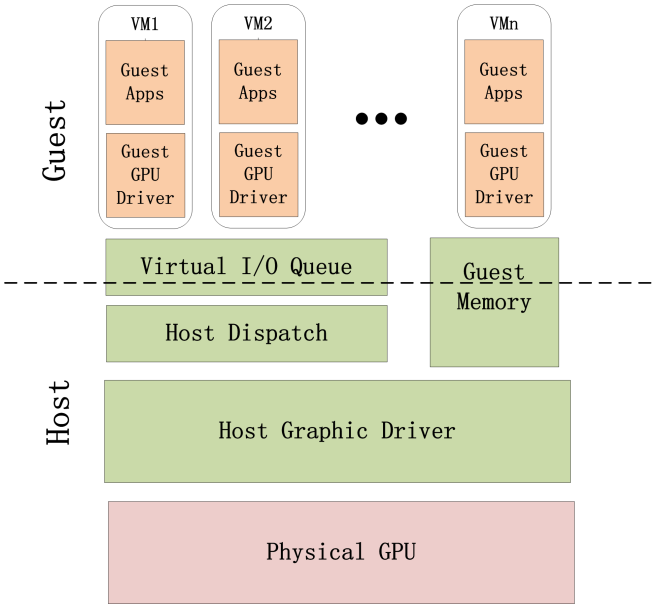


Figure 2: Cloud gaming virtualization framework

2.2. Cloud Gaming

Cloud gaming aims to provide end users frictionless and direct play-ability of games across various devices [45]. The user usually operates the game through a thin client on mobile devices. The controls from the users are transmitted to the cloud server. The graphics cards in the server rack will be responsible for rendering the game instance for each user. After that, the screen frames of the game will be captured and compressed by the video encoding unit and send to the user end through the broadband network. The user client decodes the video stream and playback on the screen. Cloud gaming allows ordinary users to access the demanding high-end games without the need to acquire a powerful game console or computer. However, cloud gaming significantly relies on broadband connection beyond 5Mbps which is not commonly available until recent years [6][7].

GPU resource management in the virtualized cloud gaming environment is not well studied. Resource sharing in the existing virtualization solutions is often poor. For example, OnLive allocates one GPU per game instance [33]. Proprietary motherboards are also used to host more GPU chips in one server. Due to the complexity of the GPU devices, hypervisors that support GPU paravirtualization achieve near-native efficiency only recently [31] [14].

As shown in Fig. 2, in a multi-user cloud gaming server, each guest application invokes a standard rendering API and the guest GPU driver prepares the corresponding objects in the guest memory and issues the GPU command packets. These packets are pushed into a virtual I/O queue which are subsequently processed by the dispatch unit in the host. Lastly, this dispatch layer sends the commands to the host graphic driver in an asynchronous manner. Objects and contents in the guest memory are transferred to the physical GPU video memory

Figure 3: Concurrent players for the top 10 games at a popular gaming site

under the control of the host driver. Multiple-users invoke multiple game instances requiring multiple virtual machines to communicate with the driver.

The popular task scheduling mechanisms such as VMWare Player [14] tend to allocate GPU resources in a first-come first-serve manner. The host GPU driver once received the request will transfer the corresponding memory objects and run the game instance on the graphic engine. For frame-based scheduling, the GPU will render one game instance for a few frames, perform a context switching and continue to render another game. For time-based scheduling, the GPU will follow the same routine but render a game for a pre-defined time slot. Whenever the GPU performs a context switching for a new game instance, the entire graphic pipeline including all the cache hierarchy is completely flushed [4][21].

3. Static Analysis

In the cloud gaming environment, it is common that multiple users are playing the same game simultaneously. For example, in many sports and action games, players often team up to play against each other. For popular single-player strategy games, it is also likely that multiple users are playing the same game (multiple copies) at the same time. Fig. 3 is a screen capture showing the real-time and peak number of users playing the top 10 games offered by a popular on-line gaming vendor "SteamPowered" [41]. The site has 6 millions active users and half of them are playing the top 10 games. About 300K users were playing the most popular game "Dota2" at the time we captured the screen. The peak players can be around 500K for this game. This means there are 500K copies of the same game being rendered at the same time in the cloud center. Today's gaming servers leverage graphic cards designed for single game so that the GPU will treat every instance of the same game as independent and render separately. However, the phenomenon in cloud gaming creates a unique opportunity for content sharing between users if multiple copies of the same game are being rendered on the same GPU.

In this section, we focus on static analysis of graphic API calls to measure duplicated data transfers from the system main memory into GPU video memory when multiple users

are rendering the same game using the same GPU. We complete our evaluation by doing a static analysis on the traces collected by OpenGL API calls. It enables us to give a quantitative analysis on the potential improvement that one part of our optimizations may achieve. We use "APITrace" to do the job [2]. APITrace is a viable tool to intercept calls to graphic driver and store their contents in a reproducible and accessible format. A trace created by APITrace contains all relevant information to reproduce the same rendering that occurred during the original execution of the targeted program as in function names, arguments, data buffers and function return values. The trace is analyzed by identifying and inspecting draw calls that would trigger the data transfer between the system memory and GPU memory that will show us how many redundant transactions can be avoided.

We aim to share the contents among different instances of the same game. There are three major types of contents that can be shared:

- The object primitives: these are the primitives (triangles, dots, lines) constructing an object model (car, tree, animal, human, etc.) in a 3D scene. Since multiple copies of the same game are being rendered, it is likely that the same objects will appear.
- The texture map: this is texture (wood, gold, water, etc) used for rendering objects. Texture maps are generally large and take up significant portion of the memory space and bandwidth. For the same reason, it is likely the texture maps can be shared.
- The shader program: this is the program instructing the graphic engine to do the rendering computation. Since it is basically the same game, it is also likely that shader programs can be shared across multiple instances.

In this paper, we focus on only sharing the texture data because it is memory hungry. The application of our technique to the object primitives and shader programs are trivial. OpenGL provides the programmers with buffer objects that can be declared with usage hints – **Static** (texture once loaded never changes); **Dynamic** (texture changes once in a while); **Stream** (texture changes frequently) [34]. It is highly possible to share static textures while the chance of sharing dynamic and stream textures is relatively low. Fortunately, the usage hints in OpenGL make life much easier to identify what to share in a game. To show how much of the texture data can be shared with another instance of the game, we sequentially went through two traces collected from two individual players, captured the contents of every data transfer and compared them for redundant occurrences. Note that although the two player were playing the same game simultaneously, the collected traces and the real-time frames displayed on their monitors can be quite different. Fig. 4 shows frame #45 of both user's screens that didn't look alike, but there were still a lot to share in the underlying GPU hardware. This is understandable because both copies of the game use the same texture and materials to render wall, floor, doors and gun, etc.

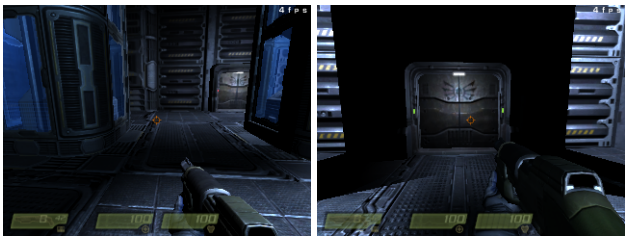


Figure 4: Quake 4 frame #45 of trace 0 and 1

These would the same contents for the same game.

	Situation	
Game	Same scenario	Different scenarios
TF2	43.95%(53MB)	40.08%(51MB)
A:TDD	27.64%(19MB)	17.31%(8MB)
UT2K4	69.66%(49MB)	53.37%(35MB)
Doom3	49.93%(55MB)	45.47%(49MB)
Q4A	39.91%(28MB)	32.39%(24MB)
Prey	51.03%(36MB)	44.98%(30MB)

Table 1: OpenGL static analysis for 6 games: "Team Fortress 2", "Amnesia: The Dark Descent", "Unreal Tournament 2004", "Doom 3", "Quake 4 Arena", "Prey"

We analyze data from six popular games each with two players. For each game we analyze two different situations:

- *Same scenario*: Two players are playing the same part/stage of a game. This is typically the case that players team up and co-play a game. For example, up to 10 users can join a single basketball match.
- *Different scenarios*: Two players are playing completely different parts/stages of a game. This is typically the case that people play the same game independently. For example, user A is playing the first stage of "Doom 3" while user B is playing the fourth stage.

Tab. 1 shows the percentage and volume of duplicate texture data transfers between the system and video memory in two seconds play time of each game. For most cases with two players, we observer a large number of redundant data transactions ranging from 30% to 70%, regardless of *same scenario* or *different scenarios*. The static analysis shows the great potential to eliminate the unnecessary memory and bus operations that can save significant portion of the system bandwidth and memory space. The only exception is in the case of "Amnesia: The Dark Descent". This game has extremely fast-changing scenes and large variance across so that the data duplication rate is low.

4. System and Architecture Changes

In order to efficiently utilize the potential benefit of content sharing, we device system and architecture modifications that can be combined to achieve the optimal results. The first set of changes affects the driver software design that controls how the GPU hardware handles memory transfers. This solution

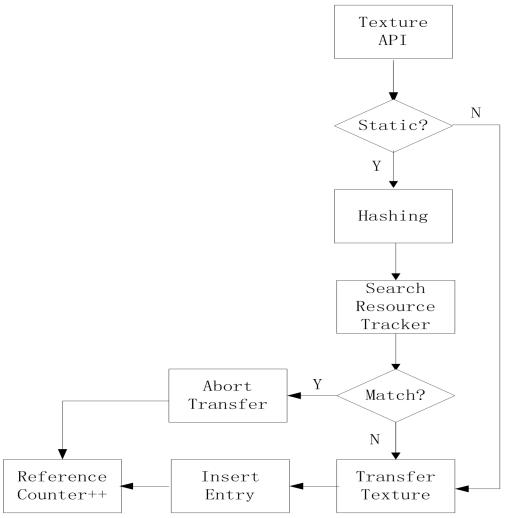


Figure 5: The program flow implemented in the driver

offers a transparent usage of the existing software stacks and games at the expense of slightly increased CPU computation. But it brings the significant advantage of saved memory transactions and space. The second change needs to change GPU cache design. This kind of hardware modifications has to be implemented by the GPU vendor. The overall driver and hardware changes are minimal.

4.1. System Change: GPU Driver Modifications

Our scheme relies on being able to identify texture maps that are duplicated. We want to save DMA transfers from the main memory to video memory and allocate less video memory space for the texture. This means we need the capability to check the data equivalence in the memory. To take advantage of the existing software stack, we propose the following driver changes to achieve the best efficiency while maintaining the system compatibility.

The way we do it is straightforward. The driver captures API calls from the virtualization layer and identifies different types of texture data through the usage hints. Direct3D uses a "buffer descriptor structure" to propagate properties of a data buffer to the GPU driver. It is used whenever a new memory buffer needs to be allocated. This structure contains a usage hint, as already described in Section 3. In OpenGL, a memory buffer can be allocated using a call to "glBufferData", which takes a target, buffer size, data and a usage hint as the inputs. In both cases, the driver can easily identify the type (texture or not) of memory object to be created and its characteristic (static, dynamic or stream).

As described in Section 3, static textures are easy to share because they are relatively stable. Dynamic and stream textures are hard to share because they vary frequently. So in our design the driver will always transfer the dynamic and stream textures while the static textures will need further processing as shown in Fig. 5

We modify the driver that is responsible for allocating/deallocating video memory and controls the memory transfers. To be more specific, we add a resource tracker into the driver that keeps track of all the texture maps having been allocated in the video memory. The tracker is basically a mapping table stored in the system memory with every entry containing a unique texture ID as the key and a video memory descriptor and a reference counter as the corresponding value pair. The memory descriptor is a pointer to the exact location in the video memory where the corresponding texture map is stored. It will be returned when the texture is being requested by the graphic engine in order to locate the texture contents in the video memory. The reference counter is used to record the number of draw calls which actively use this texture.

In the conventional method, whenever the game API attempts to load a texture, the driver will just allocate the video memory space and transfer the texture data over to the video memory. In our modified driver as shown in Fig. 5, it will first check through the resource tracker to see if it is already present in the video memory by comparing with the texture ID. If there is a match, this means another copy of the game has already loaded the same texture into the video memory and the texture content can be shared. In this case, the driver will abort the texture transfer but instead transfer the video memory descriptor to the GPU. Later if the graphic pipeline wants to render the texture, it can just use the descriptor to find the shared content. This kind of sharing can save significant amount of redundant data transfers and video memory space. The driver also increments the reference counter by one after the process.

It is important to assign a unique ID for every texture so that the driver can easily identify a match in the tracker. The way we choose is to hash the texture and obtain a hash value as the texture ID. We use the "Murmur Hash" as the hash function [28], since it is a very popular and efficient way of hashing non-cryptographic data. A match of the texture ID means the same texture has already been loaded into the video memory. The hashing function is part of the driver and takes CPU cycles. It is executed every time a static texture is requested by the game API. The hashing can potentially degrade the system performance, but we will show in Section 6 that the saved memory transfer time can well offset the hashing overhead and result in an overall system performance gain.

Whenever there is a miss in the tracker which means there is no existing matching texture in the video memory, the texture will be transferred and an entry will be inserted into the tracker. In case the amount of the video memory space is below a threshold, we need to free some memory first. The driver chooses the victim texture based on the values in the reference counters. The driver will always invalidate the texture with the smallest reference counter value which means this texture is least frequently used and discarding it may not have big impact.

5. Experiment Setup

In this section we will discuss our experiment setup. We will start by explaining our undertaken changes to the Gdev driver on a real cloud gaming server and the ATTILA GPU simulator.

5.1. Real System Setup

We build a real cloud gaming system with the configurations shown in Tab. 2. We assume two users and setup two virtual machines with each owning two cores and 2GB main memory. Ubuntu 12.04 linux OS is used as the operating system for both the host and guests. We use VMWare [14] as the paravirtualization software.

Parameter	Value
CPU	AMD Athlon X4 Quad-core 631 2.6 GHz
Memory	16G DDR3-1333 Dual-Channel
Mainboard	Biostar A55 chipset
Graphics Card	Inno3D Geforce GT440 1GB
CUDA core	96
GPU frequency	810 MHz
GPU memory	1GB GDDR5 @ 3200MHz 128bit

Table 2: Gaming System Parameters

Most of the system hacks lay in the host graphic driver layer as shown in Fig. 2. Because the driver design is the top secret for graphic card vendors like NVIDIA and AMD, we have to leverage the open source GPU driver "Gdev" [24] to evaluate the impact of our technique on a real system. Gdev is a GPU runtime and resource management engine that enables flexible GPU control and tuning. Gdev has an option to define shared memory area in the GPU which greatly facilitates our goal to allow game instances to share data and avoid unnecessary memory transfers. The interface is based on the standard SysV IPC [24].

We use software to record the real-time frames per second of each game. This is the direct reflection of the gaming performance. We also use software to record the memory transactions and memory usage. If needed, we use a multimeter to measure the current flowing through the GPU card and calculate the power consumption. Fig. 6 is a snippet of our system.

5.2. GPU Simulation Setup

Since we cannot add hardware features into the graphic card, we have to leverage a GPU simulator to exploit the architecture changes we proposed such as the enhanced cache designs. We use "ATTILA" [13], a detailed open source end-to-end graphic pipeline simulator which can report hardware status such as pipeline and cache behavior. ATTILA was originally developed to simulate a single trace. In order to simulate content sharing among multiple games, we extend the functionality to allow ATTILA to run multiple traces with a context switch

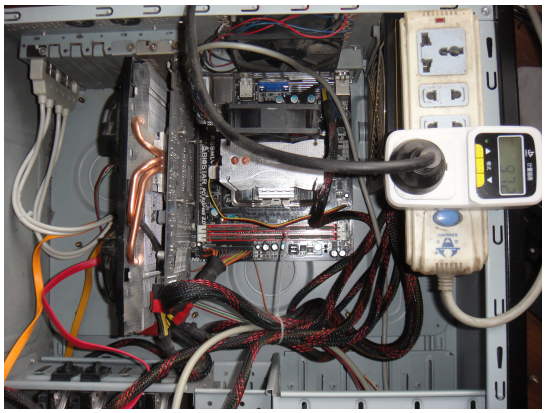


Figure 6: The cloud gaming system used for the experiments

at the end of every rendering period. The crucial ATTILA simulation parameters are listed in Tab. 3.

Parameter	Value
GPU Clock	500 MHz
Shader Clock	500 MHz
Memory Clock	500 MHz
Memory Size	256 MB
Vertex Shaders	8
Fragment Shaders	4
Streamer Unit Cache Lines	32
Streamer Unit Cache Line Size	256 Bytes
Texture Unit L0 Cache Lines	8
Texture Unit L0 Cache Line Size	64 Bytes
Texture Unit L1 Cache Lines	32
Texture Unit L1 Cache Line Size	64 Bytes
Texture Size	4*4 pixels, 64 Bytes

Table 3: ATTILA Simulation Parameters

5.3. Description of Games

We use 6 popular games in our experiment. The games cover a wide range of arena scenes, modeling styles and texture types. They are described as follows:

- *Team Fortress 2*: a team-based multi-player game developed by Valve Corporation. Team Fortress 2 is focused around two opposing teams competing for a combat-based principal objective. Players can choose to play as one of the nine classes in these teams, each with his own unique skills.
- *Amnesia: The Dark Descent*: a survival horror game by Frictional Games. The game features a protagonist exploring a dark and foreboding castle while avoiding monsters and other obstructions as well as solving puzzles.
- *Unreal Tournament 2004*: a futuristic first-person shooting game developed by Epic Games and Digital Extremes. It is part of the Unreal series, particularly the subseries started by the original Unreal Tournament.
- *Doom 3*: a science fiction game developed by id Software.

The game was a huge success with more than 3.5 million copies sold. Critics praised the game's outstanding graphics and presentation.

- *Quake 4*: the fourth title in the series of Quake. The game was developed by Raven Software. id Software supervised the development of the game as well as providing the Doom 3 engine.
- *Prey*: a first-person shooting game developed by Human Head Studios. Prey uses a heavily modified version of id Tech 4 to use portals and variable gravity to create the environments the player explores.

5.4. Experiment Method

NVIDIA Grid technology [31] currently can support two games at a time per GPU. We mimic this by allowing two instances of the same game to run under the *non-sharing* and *sharing* conditions. The *non-sharing* case is the traditional solution that the two game instance will be rendered independently. The *sharing* case utilizes the technology described in this paper and implements the augmented driver for content sharing. We also run the games under *same scenario* and *different scenarios* conditions as described in Section 3.

6. Experiment Results

In this section we will discuss the impact of content sharing in a GPU-based cloud gaming server. All the results collected in this section are from the run-time statistics of our cloud gaming system described in Section 5.1 except for the results of caches which are collected from ATTILA simulator.

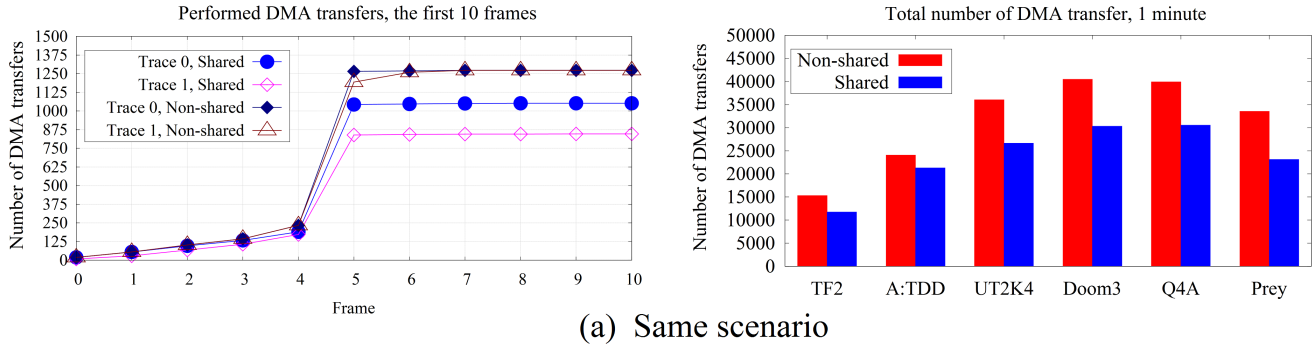
The left plot in Fig. 7(a) shows the aggregated DMA bus transfers for the first 10 frames of the game "Quake 4" under the *same scenario*. We show the two traces from two players under the *sharing* and *non-sharing* conditions. The first 5 frames are the screen loading period so that we see increasing number of static textures. The 6th frame is the first one rendering the actual arena. After that, the screen is stabilized and there are relatively less static textures transfers. Actually in many games, the majority of static textures is loaded into the video memory in the first couple of frames for rendering the arena. After this initial burst, not much static texture will be issued from the driver for some time. Obviously in case of an arena or level change, a new burst of static textures will occur and in turn call for a new wave of DMA transfers.

As shown in the left plot of Fig. 7(a), we see that the driver transfers about 20 textures (1.4MB) in the first frame with a big burst in frame 5, where 1030 textures (23MB) are transferred, In the *non-shared* case, each trace performs this workload. But for the *shared* case, we are able to reduce the DMA transfers of the trace 1 by a considerable amount. Trace 0 saves relatively less DMA transfers because it is the first running trace assigned by the driver. Trace 1 performs only 660 texture transfers in frame 5 compared with the 1030 transfers under the *non-sharing* case. This means we saved about 45% of the bus transactions for trace 1. In the right plot of Fig. 7(a), we show the long-term behaviour of all the six games. This shows the aggregated DMA transfers for one minute of game play. The game "A:TDD" doesn't perform well because the reuse rate of the static textures due to the fast change scenes in this game as discussed in Section 3.

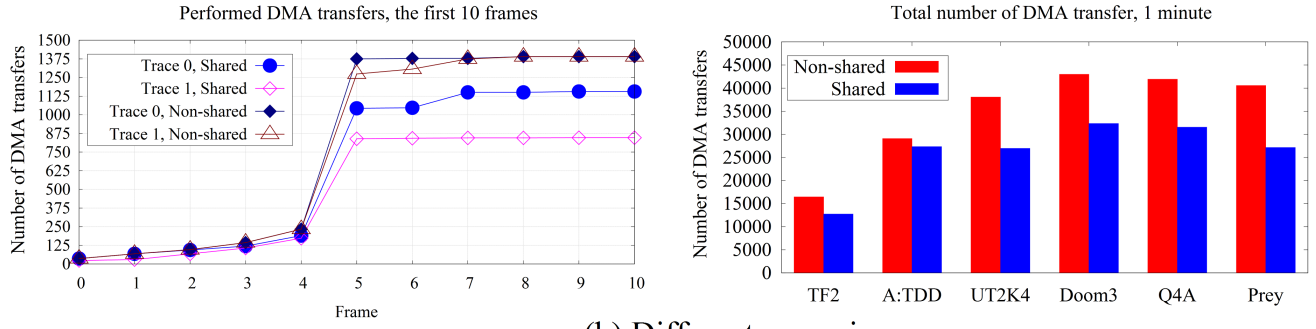
Fig. 7(b) shows the same sets of results under *different scenarios*. In this case, sharing content is combined with different gaming environment. This test is designed to reveal the true potential of content sharing in the real cloud gaming platform where multiple users are playing the same game but at different scenarios. It is obvious that in the real gaming environment, the gaming arena will keep changing due to the movement of different players. So it is imperative that we analyze the effect of content sharing under this condition. This figure again proves that the high sharing potential of textures is possible even at *different scenarios* of the same game. These results are not surprising as have been seen in Section 3. Even with different arenas, there are still a lot to sharing because the game typically uses the same textures for rendering objects such as walls, floors, water and sky. We find that on average 31.84% of DMA transfers can be eliminated.

Fig. 8 shows the allocated video memory space. The left plots are the video memory condition for the first 10 frames. We observe quite similar trend as in Fig. 7. Differently, we see relatively staggering lines for the first 5 frames. This is because the texture sizes vary quite a lot and individual frames may incur large variance in the total amount of memory needed for holding the textures. On average for the six games in our experiment, 26.89% of video memory space can be saved as shown in the right plot of Fig. 8. The capacity of the video memory is a precious resource especially under cloud gaming where multiple games divide the entire memory space.

In Fig. 9, we analyze the results from the rendering performance point of view. GPU Cycles are one of the most important factors that affect the user experience of a cloud gaming platform. Other factors include network connections, etc. We measure the real-time Frames Per Second (FPS) which is a direct reflection of the game rendering performance. An FPS rate over 30 means no flicking to the human eye but high quality games require an FPS over 60. The left plot in the figure shows the FPS for one minute play time in "Quake 4". We observe a constant trend that the *shared* case has a higher FPS than the *non-shared* case. Sharing content incurs overhead in CPU side because the CPU spends more cycles in the driver to do the hashing. But the much saved DMA transfers greatly reduce the memory transaction time so that we observe an overall net gain in the gaming performance. This is essentially important in the future cloud gaming servers when many graphics-intensive instances co-exist so that the system cannot guarantee a constant rate above 30 FPS. Using content sharing will potentially reduce the chance of "lagging or flicking" in the game play. The valleys in the plot is caused by intensive engagement between two players, such as the two players encounter for a fight. This usually imposes rendering pressure on

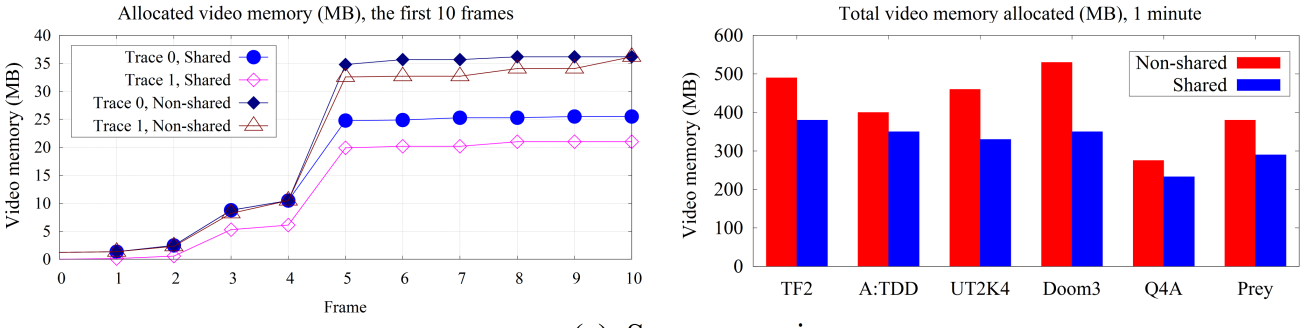


(a) Same scenario

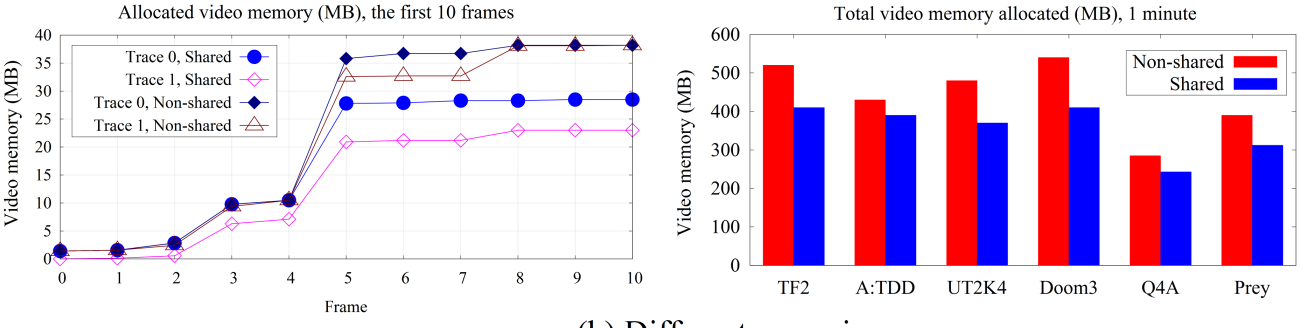


(b) Different scenarios

Figure 7: DMA transfers under different cases



(a) Same scenario



(b) Different scenarios

Figure 8: Allocated video memory under different cases

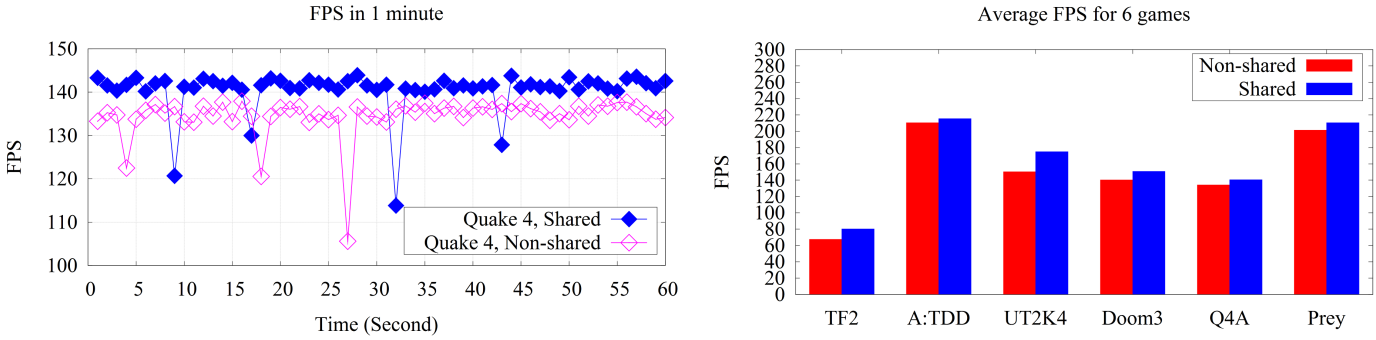


Figure 9: Frames per second under different cases

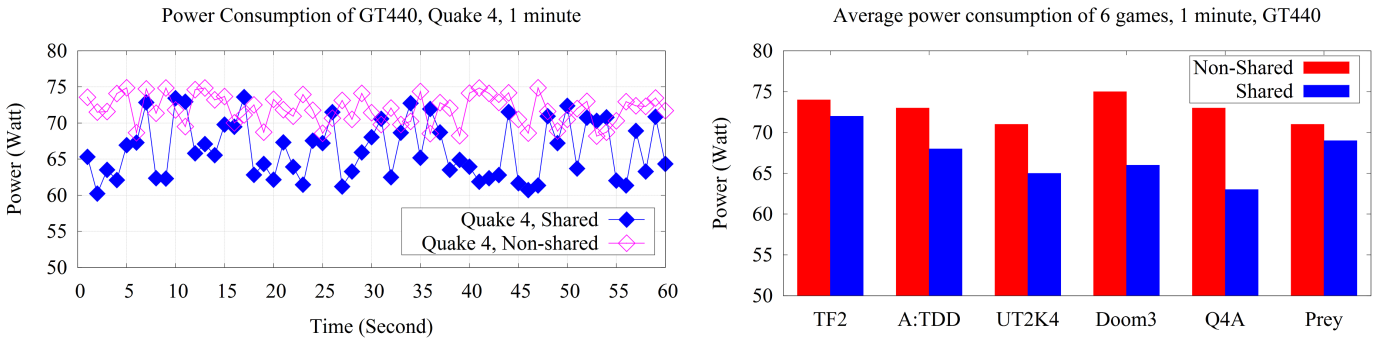


Figure 10: GPU power consumption under different cases

the GPU. For the *non-shared* case, each game instance fetches its own textures and costs memory cycles. For the *shared* case, memory cycles can be saved because of the sharing during the engagement of the players. This again proves the efficiency of our scheme. Overall, the scheme improves the average FPS of the six games by 8.99% as shown in the right plot of Fig. 9.

Fig. 10 shows the measured power results. The left plot shows the real-time power measurement for one minute in "Quake 4". Sharing content means, to some extent, the power savings since we have eliminated the redundancy. We observer a constant lower power consumption in the *shared* case. The biggest impactor here is the much reduced DMA transfers which can be power hungry. Compared with the *non-shared* case, we see large peaks and valleys in the power consumption of the *shared* case. This is understandable in that the valley usually means the texture intensive period of the game where the *shared* scheme can reduce power consumption by a lot. The peaks typically involves no texture operations so that the power consumption is comparable to the *non-shared* case. We find in an average 8.51% drop in power consumption of all the six games as is seen in the right plot of Fig. 10 which proves the better energy efficiency of our scheme.

7. Related Work

In this section, we survey the existing gaming systems and proposals for better user experiences. As far as we know, this is the first paper describing the content sharing for common game instances in a GPU-based gaming server. But there are many related works focusing on cloud gaming architecture and measurement of QoS of the cloud gaming systems. We will also introduce related works of GPU resource sharing and virtualization.

A. Cloud gaming architecture

Cloud gaming systems are generally classified into three categories: (i) 3D graphics streaming [15] [22]; (ii)video streaming [12] [20]; (iii) video streaming with post-rendering operations [39] [18].

3D graphics streaming approach aims to intercept and compress the graphics commands and stream them to the users. The user clients receive these OpenGL/Direct3D commands and then render the game scenes using its own graphics card. This approach imposes less workload on the cloud gaming server at the cost of more workload on the clients, making cloud gaming on resource-constrained devices almost impossible.

Video Streaming approach [12] [20] uses cloud servers to render the 3D graphics commands into 2D videos, compresses the video and then streams them to the target clients. The clients decode and display the corresponding video streams. This approach relieves the clients from rendering-intensive work and can be ported to different platforms.

Video streaming with post-rendering operations [39] [18] is a combination of the two methods mentioned above. Most of the rendering are performed in the cloud servers with some post-rendering such as motions, lighting and textures [9] done

on the thin clients.

B. Measuring the QoS of cloud gaming system

The Quality of Service of cloud gaming system has long been the heated topic in this area. [25] [30] give us a slow-motion analysis of an application on the server and the corresponding thin client system. However, slow-motion analysis is flawed for cloud gaming because cloud gaming services are running in real time and a slow-motion analysis can not cover all the behaviors of this service. The performance of thin client based platform has been well investigated including network traces between X-Window server and the clients under different network conditions [36]. Performance of process, memory and network bandwidth of Windows NT Terminal Service [46], as well as the performance of applications running on VNC (Virtual Network Computing) with diversified round-trip times (RTT) [43] have been well studied. Recently the performance of thin client gaming has been the popular subject of cloud gaming research. [5] proposed a methodology to study the performance of games on the general-purpose thin clients including LogMeIn [27], TeamViewer [42] and UltraVNC [44]. [8] proposed a methodology to quantify the response delay. [26] evaluated the fairness among different games under cloud gaming settings. [10] evaluated whether a wide-scale cloud gaming platform is feasible on the current network conditions and proposed a solution to lower user-end delays.

C. GPU resource sharing and virtualization

There have been some existing ideas on how to use the immense potential of GPUs for better efficiency. For instance in [24], resource-sharing protocols for the X-Window System are described to overcome a priority inversion problem for concurrent access of the same resource.

Further an API named "Gdev" [24] enables the programmer to share resources amongst contexts in a Fermi architecture GPU and deal with enhanced resource management. The main purpose is for computing applications running on GPGPU, whereas our goal lies in sharing content amongst common games and to show the potential of a possible gain. The resource management is very helpful since it enables programmers to allocate memory space exceeding the physical size of the device memory.

Other ideas focus on GPU virtualization where multiple instances try to co-execution in the GPU. Unlike the technique described in this paper where different contexts are time-multiplexing the GPU under the driver's control, the true hardware virtualization tries to achieve space-multiplexing the GPU. In [37], the open source framework "gVirtuS" [19] is extended to map many kernels on a GPU concurrently and to calculate an affinity.

8. Conclusion

In this work we are able to identify the content resources with high sharing potential in the emerging GPU-based cloud gaming servers where multiple-users can be playing the same game on the same GPU. We also calculate a static estimate on the advantages of sharing the content from different game instances. We enhance system driver design and develop CCP cache replacement policy for the texture cache architecture to further exploit the potential of content sharing GPU in the cloud-gaming environment. In our research it becomes clear that sharing textures and other memory objects brings the major advantages in the reduced memory transactions, memory space, power consumption and improved cache hit rate. We also find that the gaming performance has been improved because of the average reduced latency for memory access. Our experiment on the real cloud gaming system and the GPU simulator proves that the proposed techniques can bring significant advantage when comparing with traditional solutions.

In our future work, instead of just sharing the static texture, we will be looking into more efficient ways to share dynamic and stream textures, probably collaborating with the task scheduler. It is also trivial to share object primitives and shader programs following the same scheme. Another interesting question in the future research would be how to share computation. Since multiple players are essentially playing the same game, it is possible to even share some computation besides the memory contents. Sharing computation can greatly reduce the total GPU power consumption and at the same time boost the graphic rendering speed.

References

[1] AMD, "Radeon™ sky series announcement," 2013, [Accessed 2013-05-18]. Available: http://www.amd.com/us/products/desktop/workstation/cloud/Pages/cloud-gaming.aspx

[2] APITrace. Available: http://apitrace.github.io/

[3] ATI, "Graphics drivers: Leading the way to windows vista™," 2006. Available: http://www.ati.com/products/wp/ATIWDDMWhitepaperFinalV38.pdf

[4] I. Buck, G. Humphreys, and P. Hanrahan, "Tracking graphics state for networked rendering," in *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 2000.

[5] Y.-C. Chang, P.-H. Tseng, K.-T. Chen, and C.-L. Lei, "Understanding the performance of thin-client gaming," in *Communications Quality and Reliability (CQR), 2011 IEEE International Workshop Technical Committee on*. IEEE, 2011, pp. 1–6.

[6] K.-T. Chen, Y.-C. Chang, H.-J. Hsu, D.-Y. Chen, C.-Y. Huang, and C.-H. Hsu, "On the quality of service of cloud gaming systems," *IEEE Transactions on Multimedia*, 2013.

[7] K.-T. Chen, Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. Lei, "Measuring the latency of cloud gaming systems," in *Proceedings of ACM Multimedia 2011*, Nov 2011.

[8] K.-T. Chen, Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. Lei, "Measuring the latency of cloud gaming systems," in *Proceedings of the 19th ACM international conference on Multimedia*. ACM, 2011, pp. 1269–1272.

[9] Y.-C. Chen, C.-F. Chang, and W.-C. Ma, "Asynchronous rendering," in *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '10. New York, NY, USA: ACM, 2010, pp. 16:1–16:1. Available: http://doi.acm.org/10.1145/1730804.1730988

[10] S. Choy, B. Wong, G. Simon, and C. Rosenberg, "The brewing storm in cloud gaming: A measurement study on cloud to end-user latency," in *Network and Systems Support for Games (NetGames), 2012 11th Annual Workshop on*. IEEE, 2012, pp. 1–6.

[11] A. Cunningham, "But can it stream crysis? nvidia's new cloud gaming server explained," 2013. Available: http://arstechnica.com/gaming/2013/01/but-can-it-stream-crysis-nvidias-new-cloud-gaming-server-explained/

[12] D. De Winter, P. Simoens, L. Deboosere, F. De Turck, J. Moreau, B. Dhoedt, and P. Demeester, "A hybrid thin-client protocol for multimedia streaming and interactive gaming applications," in *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*. ACM, 2006, p. 15.

[13] V. M. Del Barrio, C. González, J. Roca, A. Fernández, and E. Espasa, "Attila: a cycle-level execution-driven simulator for modern gpu architectures," in *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 231–241.

[14] M. Dowty and J. Sugerman, "Gpu virtualization on vmware's hosted i/o architecture," *Operating Systems Review*, vol. 43, no. 3, pp. 73–82, 2009.

[15] P. Eisert and P. Fechteler, "Low delay streaming of computer graphics," in *Image Processing, 2008. ICIP 2008. 15th IEEE International Conference on*. IEEE, 2008, pp. 2704–2707.

[16] I. Gaikai, "Gaikai: 200+ titles, 50m+ gamers per month. a guinness world record," 2013. Available: http://www.gaikai.com/

[17] GamingAnywhere.org, "Gaminganywhere," 2013. Available: http://gaminganywhere.org/index.html

[18] F. Giesen, R. Schnabel, and R. Klein, "Augmented compression for server-side rendering." in *VMV*, 2008, pp. 207–216.

[19] Giunta, Giunta, Montella, Agrillo, and Coviello, *Euro-Par 2010 - Parallel Processing*, vol. 6271. Available: http://sfx.kobv.de/sfx_tub?sid=google&auinit=G&aulast=Giunta&atitle=A%20GPGPU%20transparent%20virtualization%20component%20for%20high%20

[20] O.-I. Holthe, O. Mogstad, and L. Ronningen, "Geelix livegames: Remote playing of video games," in *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*. IEEE, 2009, pp. 1–2.

[21] H. Igehy, G. Stoll, and P. Hanrahan, "The design of a parallel graphics interface," in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998.

[22] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J.-P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä *et al.*, "Platform for distributed 3d gaming," *International Journal of Computer Games Technology*, vol. 2009, p. 1, 2009.

[23] D. Kanter, "Inside nehalem: Intel's future processor and system," 2008. Available: http://www.realworldtech.com/nehalem/3/

[24] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class gpu resource management in the operating system," in *USENIX ATC*, vol. 12, 2012, pp. 37–37.

[25] A. M. Lai and J. Nieh, "On the performance of wide-area thin-client computing," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 2, pp. 175–209, 2006.

[26] Y.-T. Lee, K.-T. Chen, H.-I. Su, and C.-L. Lei, "Are all games equally cloud-gaming-friendly? an electromyographic approach," in *Network and Systems Support for Games (NetGames), 2012 11th Annual Workshop on*. IEEE, 2012, pp. 1–6.

[27] I. LogMeIn, "Logmein," 2013, [Accessed 2013-11-21]. Available: https://secure.logmein.com/CN/

[28] H. E. Michail, A. P. Kakarountas, A. S. Milidonis, and C. E. Goutis, "A top-down design methodology for ultrahigh-performance hashing cores," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 4, pp. 255–268, 2009.

[29] Microsoft, "Programming guide for hlsl," March 2013, [Accessed 2013-05-21]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509635(v=vs.85).aspx

[30] J. Nieh, S. J. Yang, and N. Novik, "Measuring thin-client performance using slow-motion benchmarking," *ACM Transactions on Computer Systems (TOCS)*, vol. 21, no. 1, pp. 87–115, 2003.

[31] NVIDIA, "Nvidia grid," 2013, [Accessed 2013-05-18]. Available: http://www.nvidia.com/object/cloud-gaming.html

[32] S. R. (NVIDIA), "Fundamental optimizations global memory," 2011. Available: http://www.stanford.edu/dept/ICME/docs/seminars/Rennich-2011-04-25.pdf

[33] OnLive, "Onlive: The leader in cloud gaming," 2013. Available: http://www.onlive.com/

[34] OpenGL.org, "Buffer object," February 2013, [Accessed 2013-05-12]. Available: http://www.opengl.org/wiki_132/index.php?title=Buffer_Object&oldid=8688

[35] K. Packard, "Gem - the graphics execution manager," May 2008, [Accessed 2013-05-21]. Available: http://lwn.net/Articles/283798/

[36] K. Packard and J. Gettys, "X window system network performance." in *USENIX Annual Technical Conference, FREENIX Track*, 2003, pp. 207–218.

[37] V. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting gpu sharing in cloud environments with a transparent runtime consolidation framework," *HPDC*, pp. 217–228, 2011.

[38] Z. Rusin, "Gallium3d: Graphics done right," 2008. Available: http://akademy2008.kde.org/conference/slides/zack-akademy2008.pdf

[39] S. Shi, C.-H. Hsu, K. Nahrstedt, and R. Campbell, "Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming," in *Proceedings of the 19th ACM international conference on Multimedia*. ACM, 2011, pp. 103–112.

[40] E. software association, "Industry facts," 2012, [Accessed 2013-11-21]. Available: http://www.theesa.com/facts/

[41] Steampowered, "Real-time stats for online users," 2013. Available: http://store.steampowered.com/stats/

[42] TeamViewer, "Teamviewer," 2013, [Accessed 2013-11-21]. Available: http://www.teamviewer.com/zhCN/index.aspx

[43] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying interactive user experience on thin clients," *Computer*, vol. 39, no. 3, pp. 46–52, 2006.

[44] UltraVNC, "Ultravnc," 2008, [Accessed 2013-11-21]. Available: http://www.uvnc.com/

[45] Wikipedia, "Cloud gaming," 2013. Available: http://en.wikipedia.org/wiki/Cloud_gaming

[46] A. Y.-l. Wong and M. I. Seltzer, "Evaluating windows nt terminal server performance," pp. 145–154, 1999.