ShareRender: Bypassing GPU Virtualization to Enable Fine-grained Resource Sharing for Cloud Gaming

Wei Zhang[†] Xiaofei Liao[†¶] Peng Li[‡] Hai Jin[†] Li Lin[†]
[†] Service Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[‡] School of Computer Science and Engineering, The University of Aizu, Japan
alanzw@hust.edu.cn,xfliao@hust.edu.cn,pengli@u-aizu.ac.jp,hjin@hust.edu.cn,llin@hust.edu.cn

ABSTRACT

Cloud gaming is promising to provide high-quality game services by outsourcing game execution to cloud so that users can access games via thin clients (e.g., smartphones or tablets). However, existing cloud gaming systems suffer from low GPU utilization in the virtualized environment. Moreover, GPU resources are scheduled in units of *virtual machines* (VMs) and this kind of coarse-grained scheduling at the VM-level fails to fully exploit GPU processing capacity. In this paper, we present ShareRender, a cloud gaming system that offloads graphics workloads within VMs directly to GPUs, bypassing GPU virtualization. For each game running in a VM, ShareRender starts a graphics wrapper to intercept frame rendering requests and assign them to render agents responsible for frame rendering on GPUs. Thanks to the flexible workload assignment among multiple render agents, ShareRender enables fine-grained resource sharing at the frame-level to significantly improve GPU utilization. Furthermore, we design an online algorithm to determine workload assignment and migration of render agents, which considers the tradeoff between minimizing the number of active server and low agent migration cost. We conduct experiments on real deployment and trace-driven simulations to evaluate the performance of ShareRender under different system settings. The results show that ShareRender outperforms the existing video-streaming-based cloud gaming system by over 4 times.

KEYWORDS

cloud gaming; GPU; fine-grained; scheduling

1 INTRODUCTION

Modern games usually involve intensive CPU and GPU computing, which cannot be afforded by mobile devices (e.g., smartphones and tablets) and even PCs without high-end hardware. Cloud becomes a perfect platform for game services thanks to its virtually unlimited hardware resources that can be accessed over the Internet. By outsourcing game programs to cloud, cloud gaming services render the interactive games remotely in the cloud and send the scenes as video or graphics streams back to the thin clients [8, 24].

Achieving good scalability and maintaining acceptable game experiences (also called Quality of Experience, QoE) simultaneously are critical for cloud gaming. Traditionally, cloud game services are deployed in *virtual machines* (VMs) [11]. They normally contain three types of computation intensive operations, including logical computing (generally on CPU), high definition 3D rendering (generally on GPU), and video encoding (on CPU or GPU), which affect the QoE and scalability greatly. Therefore, exploiting GPU parallelism becomes a key and important challenge for cloud gaming. The research of GPU virtualization is still ongoing to pursue high scalability and low cost. On the one hand, hardware-based virtualization techniques, such as NVIDIA GRID [5], provide high performance and acceptable scalability for graphics intensive workloads, but are costly for massive deployment. On the other hand, existing software-based GPU virtualization techniques [26, 29] can improve the scalability of virtualized rendering instances via optimizations on memory mapping, but they lack of abstractions of some important features of GPU [12, 22, 23] (for example, hardware acceleration for video encoding). This would cause heavy workloads on CPU, accordingly low scalability of cloud gaming. The reason is the coarse-grained resource sharing and scheduling in units of VMs, which will be shown in following motivation part.

In this paper, we propose ShareRender[1], a novel cloud gaming system that fully exploits powerful capability of GPUs, including the new features of accelerating, with good scalability and low cost. A key concept of ShareRender's design is the decoupling of GPU rendering from VMs. That is, GPU rendering function is extracted from VM, so that it can run independently on physical machines. In such a way, CPU and GPU resources can be scheduled individually, instead of being bundled together in traditional VM-based cloud gaming systems, to increase resource utilization.

ShareRender facilitates this decoupling with a lightweight and efficient design. Specifically, for each game instance running in a virtual machine, ShareRender starts a daemon, called graphic wrapper, which intercepts frame rendering requests issued from the game. Meanwhile, several render

¶ Corresponding Author.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MM '17, October 23–27, 2017, Mountain View, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN ISBN 978-1-4503-4906-2/17/10. . . $15.00
https://doi.org/10.1145/3123266.3123306

[1]ShareRender, https://github.com/CGCL-codes/ShareRender

agents are started on different physical machines. They receive rendering requests from the associated graphic wrapper and conduct frame rendering. Note that render agents interact directly with GPUs via the graphic driver, eliminating the involvement of the GPU virtualization.

To guarantee the correctness and efficiency, we further address the following design challenges in ShareRender. First, each graphic wrapper encapsulates rendering requests as graphic tasks and can dispatch them to multiple render agents. The statuses of render agents should be consistent with graphic wrapper to ensure correct rendering results. We design a context synchronization mechanism that synchronizes statuses of render agents such that they can collaborate on rendering tasks.

Second, to feed geometric data to render agents for frame rendering, we need to frequently retrieve data from GPUs, which is a slow process and incurs large overhead. We propose to create shadow objects in main memory for geometric objects such as index buffer, vertex buffer, and texture. When graphic wrappers need to load geometric data, they can quickly fetch them from main memory, instead of accessing the slow GPU memory.

Finally, we find that game workloads are dynamic. They are determined by user operations in games and cannot be accurately predicted, which is also confirmed by [20]. The dynamic workloads may incur GPU overload. A simple solution is to create multiple render agents and dynamically adjust the amount of workloads assigned to them. However, it is still with limited flexibility and cannot fundamentally solve the overloading problem. To deal with this challenge, we design a migration mechanism that can migrate render agents between different physical machines to increase the flexibility of resource sharing. We also propose an online algorithm to decide how to assign graphics tasks among render agents, and when render agents should be migrated with the consideration of making a tradeoff between minimizing the number of active servers and low agent migration overhead. This algorithm has low computational complexity and can be easily implemented in practice.

The main contributions are summarized as follows.

- We develop a cloud gaming system called ShareRender, based on our previous work [18], with a series of novel designs, such as context synchronization, shadow objects, and render agent migration. ShareRender is able to achieve high efficiency and correctness of frame rendering in a distributed environment.
- We conduct experiments on real deployment and trace-driven simulations to evaluate the performance of ShareRender under different system settings. The results show that ShareRender significantly outperforms existing cloud gaming systems.

The rest of the paper is organized as follows: Section II motivates our design by experiments. Section III presents the system overview. Section IV describes our system design. Section V presents evaluation results. Related works is included in section VI. Section VII concludes this paper.

2 MOTIVATION

To motivate our design, we conduct experiments on a well-known open-source cloud gaming system, called GamingAnyWhere (GA) [13], deployed in VMWare virtual machines. We set GA to work in the periodic mode in which it captures GDI (*Graphic Device Interface*) window area periodically and applies H.264 to encode frame data for video streaming. All experiments are conducted on a server equipped with Intel i5 3.3GHz CPU and NVIDIA Quadro 2000 GPU.

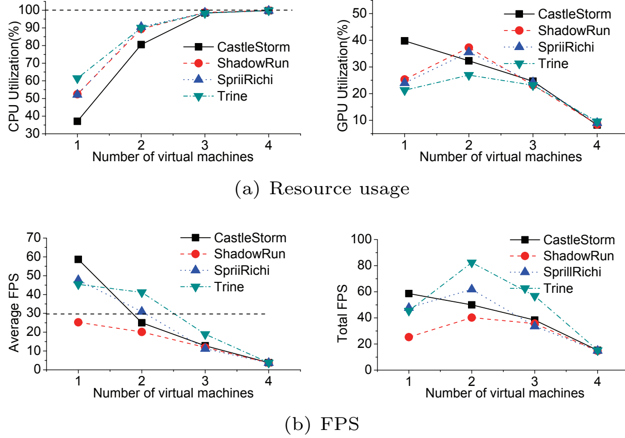
2.1 Resource Utilization in Cloud Gaming

We consider 4 types of games, as selected by LiveRender [18], with different GPU/CPU utilizations, including CastleStorm, ShadowRun, SprillRichi, and Trine. We start several game instances for each game, each of which runs in a dedicated virtual machine. We measure both CPU and GPU utilization that indicates the portion of hardware resources used by the cloud gaming system. As shown in Fig. 1, CPU utilization grows to 100% as we increase the number of concurrent game instances from 1 to 4. In contrast, GPU utilization reaches the ceiling of 40% when there is one instance of CastleStorm or two instances of other games. For games except CastleStorm, only one instance cannot saturate GPU resources, so the utilization grows when we start two instances. However, further increasing number of instances decreases GPU utilization. The main reason is that VMs have unfriendly support for hardware acceleration, makes GA push heavy load (frame capturing, format conversion, and video encoding) on CPU. As a result, CPU resources are quickly exhausted and become the performance bottleneck. Although there are unoccupied GPU resources, they cannot be used by games, leading to low GPU utilization.

We also measure the FPS (*frames per second*), reflecting QoE. Normally 28 FPS is enough to guarantee smooth game playing [31]. The total FPS of all concurrent game instances is illustrated in Fig. 1(b), where the curves of all games show similar trend with their GPU utilization. However, the FPS per instance decreases under all games as the number of concurrent instances grows. In particular, even though GPU utilization and total FPS increase when two instances of ShadowRun are started, GA cannot guarantee smooth game playing because each instance has less than 28 FPS. The reasons may be resource contention among multiple game instances [31] and large overhead of retrieving data from GPUs [1] in VM, especially retrieving 2D graphic data using GDI. The quality of all games becomes worse as more instances are started. Above experiments demonstrate that existing cloud gaming system cannot fully exploit GPU resources in a virtualized environment.

2.2 GPU Workload Dynamics

We run two game instances of Trine in two VMs, respectively, and show the hardware utilization as well as their FPS during 350 seconds in Fig. 2. In the beginning, both game instances have smooth playing with above 30 FPS. When one game instance, whose curve is denoted by "Trine 1" in



(a) Resource usage

(b) FPS

Figure 1: Experiment results of GamingAnyWhere

Fig. 2, enters a new scene after 30 seconds, its GPU resource demands suddenly increase, leading to severe resource contention that lowers GPU utilization. As a result, FPS of both game instances drops below 30. Later, "Trine 1" is stable at about 40 FPS, but "Trine 2" fluctuates due to frequent scene changes. The dynamic of GPU utilization is consistent with total FPS of both game instances.

The experimental results in Fig. 2 motivate us to consider the game workload dynamic in the system design. An intuitive method is to migrate VMs to ease hardware resource contention. However, VM migration is a slow process involving a large amount of data copy over the network. Therefore, we need a lightweight workload migration scheme.

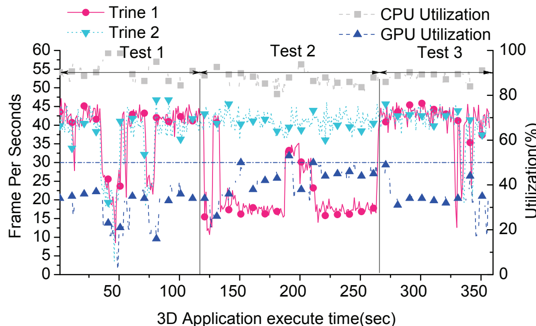


Figure 2: Overload in video streaming

3 SYSTEM OVERVIEW

ShareRender is a novel cloud gaming system based on video streaming. It consists of a thin client and three main components: graphic wrapper, render agent, and scheduler, which are integrated into the virtualized environment at the cloud. As shown in Fig. 3, the thin client at the user side captures user inputs and sends game requests to cloud. Meanwhile, it is responsible for game playing after receiving the video stream from cloud. For each game requested by users, we start a dedicated VM to run a game instance in the cloud. Within each VM, a graphic wrapper resides between the game instance and guest operating system. It intercepts 3D graphic APIs (e.g., Direct3D and OpenGL) issued by the game instance and encapsulates them as graphic tasks by including associated data.

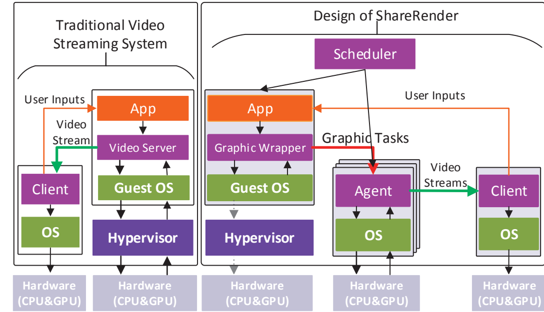


Figure 3: System architecture of ShareRender

On each physical machine, we start render agents on the host operating system, which are responsible for running graphic tasks on GPUs and encoding result frames into video streams. We can create multiple render agents for a single game instance, and they are distributed to different machines. As a result, a graphics wrapper can send its graphic tasks to any render agent associated with the same game instance. Furthermore, render agents can be migrated among different physical machines. Compared with VM migration, a render agent can be quickly migrated with low overhead because of its small size.

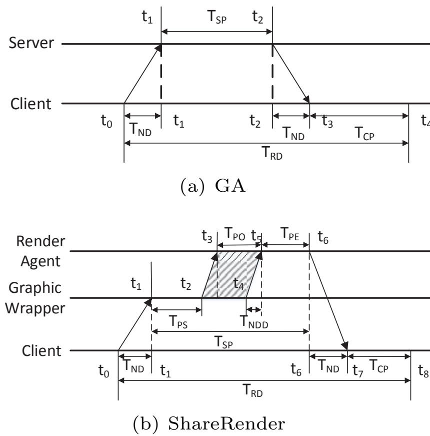
A *scheduler* is developed to optimize the workload assignment on GPUs. It periodically collects system information without interrupting game services, such as loads of graphic tasks and GPU utilization, and decides where render agents should be migrated to, and how to assign graphic tasks to render agents. Our objective is to pack graphic tasks into minimum number of servers to reduce energy consumption. Meanwhile, we consider to avoid frequent render agent migration to lower system overhead.

For clarity, we show the system processing in Fig. 4, which clearly illustrates the interactions between different components of GA and ShareRender. In GA, a client's input events are captured and then transmitted to the server after T_{ND} (network delay) time. The server also spends T_{ND} time to send video stream back to client. The server spends T_{SP} (server processing) to process the input, render the graphics, and finally generate a video stream. It takes the client T_{CP} (client processing) to decode and display the scene on screen. The total response delay is denoted by T_{RD} (response delay).

ShareRender has a more complex process. After receiving client's input, the graphic wrapper starts a pipeline to generate graphic tasks and dispatch them to render agents. The pipeline start time at graphic wrapper is T_{PS} (pipeline starting) and pipeline end time at render agent is T_{PE} (pipeline ending), the transmission of graphic tasks is in parallel with task generation, for a period of T_{PO} (pipeline overlap). The graphic tasks are transmitted over the data center network to render agent with a delay of T_{NDD} (network delay in datacenter). After that, the render agent processes the graphic tasks, and sends the video stream directly back to the client.

4 SYSTEM DESIGN

In this section, we present the design details of main components including the graphic wrapper, the render agent, the *scheduler*, and the client.



(a) GA

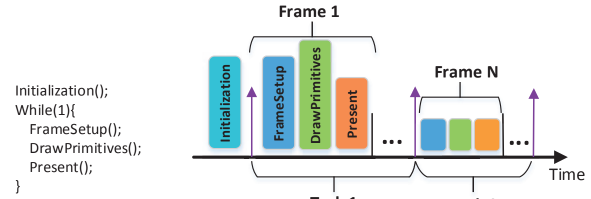
(b) ShareRender

Figure 4: Response delay of GA and ShareRender

4.1 Graphic Wrapper

An important function of the graphic wrapper is to divide GPU rendering workloads into graphic tasks in units of frames for fine-grained scheduling. It is highly related with the GPU computation model, as shown in Fig. 5(a). Games initialize the GPU and then enter an infinite loop, in which each iteration is in charge of rendering a frame. For each frame's rendering, games first call *FrameSetup()* to setup rendering pipelines, including lights settings to illuminate objects in scene, transformation settings for geometry data, etc. Then the function *DrawPrimitives()* is invoked to render all geometry objects in the frame. Finally, the function *Present()* is invoked to flush all commands to GPU device and output the results. According to the above discussion, ShareRender is able to generate basic graphic tasks as illustrated in Fig. 5(b) (the figure shows the concept: one frame, one graphic task. In our system, a fixed number of frames are grouped into one graphic task). When a graphic task is dispatched to a render agent, the agent needs to: 1) initialize the GPU as the function *Initialization()* in Fig. 5(a); 2) prepare the special settings for each frame rendering as *FrameSetup()*; 3) locate and transfer the data for geometry objects needed by the *DrawPrimitives()*. A game context is used to describe the current game status on GPU, which contains the data and operations for GPU initialization, frame settings for rendering pipeline, geometric objects, etc. The game context is very important to ensure the correctness for basic graphic tasks to be rendered in different render agents.

There are two challenges based on above analysis. First, the important context data is stored in graphic wrapper and may not be transmitted to the render agents when the wrapper dispatches graphic tasks. Therefore, we study how to synchronize the context data with high efficiency. Another challenge lies in geometric data transmission. Game instances usually push important geometric data to GPUs. Since graphic wrappers work at the API level, they are unaware of the semantics of these geometric data in games and how games update them. When graphic wrappers prepare graphic tasks, they need to frequently retrieve geometric data from GPU memory to ensure correctness, which incurs large overhead because reading data from GPU is slow [4].



(a) Pseudocode (b) Illustration of graphic task

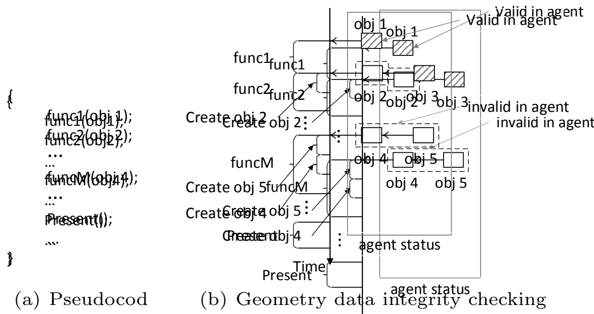
Figure 5: Computing model of games and example of graphic tasks in ShareRender

Therefore, we design a context synchronization mechanism to guarantee the correctness of rendering, which will be presented in Section 4.1.1. Moreover, we create shadow geometric objects in graphic wrapper in the main memory to eliminate slow data access to GPU memory, which will be presented in Section 4.1.2. Refer to source codes for other implementation details.

4.1.1 Context Synchronization. Each render agent maintains a game context, which is synchronized with the one in its graphic wrapper for correct rendering. Note that only the agent having graphic tasks synchronizes the current game context, while other agents without tasks do not need context synchronization. Since the consecutive frames usually show strong similarity [18], only a small portion of the context data will be updated and synchronized when new frames arrive. Here, we design two mechanisms: *sequence recorder* and *geometry data integrity checking*, to guarantee the correctness and efficiency of context synchronization. They are elaborated as follows.

Sequence recorder: Sequence recorder records all graphic operations related to the GPU initialization and each frame's special settings as well as their parameters dependency in the game context. These operations are synchronized to render agents and are applied in the same sequence to generate correct settings of rendering pipeline. As GPU initialization operations have no explicit beginning and ending, we record them from the beginning of GPU initialization until the first geometric object is created. For those operations invoked in *FrameSetup()*, we record them when they are defined. When a graphic task is dispatched to a new agent, the graphic wrapper first sends the recorded initialization operations to the selected render agent. For each frame, the graphic wrapper sends necessary initialization operations for operations in *FrameSetup()*, which are not defined in the graphic task.

Geometry data integrity checking: Graphic wrapper needs to send graphic operations as well as their parameters to render agents during *DrawPrimitives()*. Since the parameters usually contains only references to geometry objects, it is necessary to check data integrity for geometry objects in the render agent's game context. *Geometry data integrity checking* decides which geometric objects should be sent to guarantee geometry objects' integrity and to avoid redundant data transmission. If an object already exists in the game context of a render agent, it is unnecessary to send it again. As an example shown in Fig. 6, the graphic wrapper will not send the geometric object *obj1* because it already exists



(a) Pseudocod (b) Geometry data integrity checking

Figure 6: Context synchronization example. ***func1*** **has a parameter refer to** ***obj1*****,** ***func2*** **has a parameter refer to** ***obj2*****. If** ***obj2*** **is invalid in render agent, graphic wrapper creates** ***obj2*** **before sending** ***func2*** **to render agent. If both** ***obj4*** **and** ***obj5*** **in** ***funcM*** **are invalid in render agent, they are created recursively.**

in the render agent. A more complicated case is that some geometric objects form a dependency chain. For example, a surface is always created from a texture, and the texture depends on a correct render pipeline status. *Geometry data integrity checking* recursively checks the objects along the chain and sends the missing ones to render agents. In Fig. 6, *obj2* and *obj3* are in the same dependency chain. If *obj2* is missing, graphic wrapper sends it to the render agent.

4.1.2 Shadow Objects. We create a shadow object in main memory for each geometric object such as index buffer, vertex buffer, or texture, so that when graphic wrappers need to load geometric data, they can quickly obtain them in main memory, instead of accessing the slow GPU memory. The graphic wrapper monitors a pair of operations: *Lock* and *Unlock*, on geometric data. The geometric data access starts with *Lock* and finishes with *Unlock*. As shown in Fig. 7, after the *Lock* operation being issued by the game application, we redirect all geometric data access from games to shadow objects in main memory. When the game conducts *Unlock* operation, we write the updates of shadow objects back to their counterparts in GPU memory using a similar pair of *Lock* and *Unlock* operations. Meanwhile, these updates are synchronized to the active render agent.

Additionally, the challenge in designing shadow object lies in satisfying various demands by different types of geometric objects. For example, shadow objects of textures require to handle mipmaps [7] and the relationship between textures and their mipmaps, while shadow objects of vertices require to minimize the size of updated data. For texture, we create shadow objects for all mipmaps inside a texture and add the textures to their mipmaps' dependency list. For vertices, we cache the shadow objects recently updated and update only the different data compared to the cached one. According to our experimental results, shadow objects can accelerate data access by 10x with limited additional memory occupation.

4.2 Render Agent

Each render agent contains a graphic replayer and a video module. The graphic replayer extracts rendering commands and parameters, and renders game scenes based on local

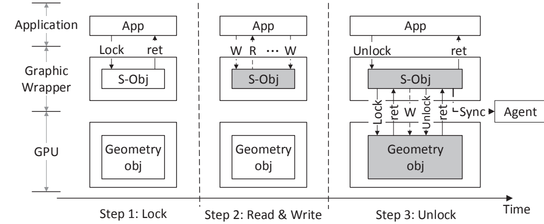


Figure 7: Illustration of shadow object access

context. When graphic replayer flushes graphic calls, usually by invoking *Present()*, video module encodes rendered frame data into video streams. Either CPU encoder or GPU encoder can be used by the video module.

In ShareRender, migration of a render agent involves terminating the current render agent and creating a new one at another physical server designated by the *scheduler*. After a render agent is created, it synchronizes game context with its graphic wrapper. ShareRender implements a pre-copy-like live migration mechanism but without down time, and the duration is much smaller compared to the duration of VM live migration. Multiple render agents share hardware resources according to the strategy given by graphic driver, while the workloads assigned to each render agent is determined by graphic wrapper.

4.3 Scheduler

The *scheduler* is responsible for optimizing the assignment of graphic tasks on GPU. We propose an online heuristic algorithm to make decisions without the knowledge of future workloads. Specifically, we run the *scheduler* periodically and each game instances has at most K render agents. At the beginning of each period, we estimate the graphic task workloads in terms of occupied GPU resources in the current period. To reduce the number of active physical machines running graphic tasks, we follow a similar principle with classical bin packing algorithm, which always selects the most suitable server that can accommodate the graphic task. However, the most suitable server may have no render agent for this task, leading to agent migration. To control the agent migration overhead, we prefer to assign a graphic task to the server containing a render agent. If these candidate servers with render agents have insufficient resources to accommodate this task, we migrate the agent to another server with enough resources and dispatch the graphic task to it.

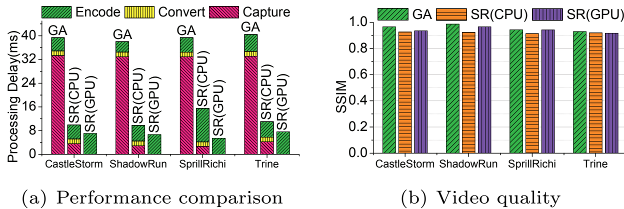
4.4 Client

The client of ShareRender is designed for computers and devices with limited hardware resources, such as smartphones, tablets. We use portable libraries to implement client to provide strong adaptability. It has two main functions: input handling and game video replay. The user input events are captured via SDL [3] and sent to graphic wrapper in the cloud. We implement a RTSP [6] client via live555 [2] library to deal with video and audio streaming. The RTSP client responses to multiple video streams and manages the switch of frame data source.

5 EVALUATION

We build a virtualized environment based on VMWare Workstation 12.0 installed on a server equipped with an Intel i5 3.3GHZ processor, 16GB DDR3 memory, a NVIDIA Quadro 2000 GPU with 1GB GDDR5 dedicated memory. We conduct experiments to compare ShareRender with another video-streaming-based cloud gaming system: GamingAnyWhere (GA), by running four popular games: SprillRichi, CastleStorm, Trine, and ShadowRun. We also collect traces generated by experiments, and use them to conduct large-scale simulations to evaluate the performance of ShareRender under different system settings.

5.1 Video Performance and Video Quality



(a) Performance comparison (b) Video quality

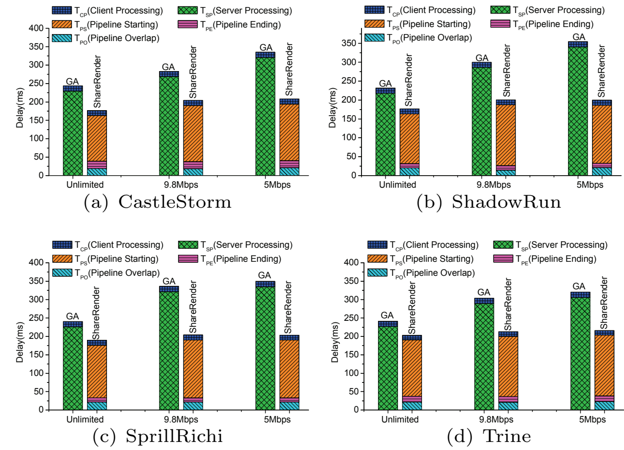
Figure 8: Performance comparison and video quality of games when using CPU encoding and GPU encoding, SR represents ShareRender.

Generating video streams is compute-intensive task and the implementation of video module can directly affect system performance of video streaming based cloud gaming systems. Both GA and ShareRender generate video streams with three phrases: capturing, format conversion, and encoding. GA can only use CPU encoding without hardware support, while ShareRender has flexibility in adopting either CPU or GPU encoding because of the benefits of render agents. We insert the code to measure the processing delay in each phase and setup both GA and ShareRender on physical machine. As shown in Fig. 8(a), ShareRender can significantly reduce the time on capturing, especially in GPU encoding mode, leading to less video generation time compared to GA.

We measure the video quality at clients. We use the X264 codec for video streaming as GA does, and also adopt CUDA to acceleration the encoding processing. We set the frame rate of 30 FPS with an upper limit for the bitrate. For each game instance, we select 40 frames of typical game scenes at server side and capture the decoded frames of the same scenes at client. We evaluate their structural similarity (SSIM) and use the average value as the final results of video quality. As shown in Fig. 8(b), ShareRender achieves high SSIM close to GA. Their performance gap is less than 6.39% for CPU encoding and 3.21% for GPU encoding, respectively, but both achieve high quality standard. ShareRender achieves good balance in performance and high quality.

5.2 Response Delay

Response delay (RD) is critical for gaming experience. As described in Section 3, response delay is defined as the interval between the time when an input event is triggered in client and the time when the resulting game scene can be seen. The



(a) CastleStorm (b) ShadowRun (c) SprillRichi (d) Trine

Figure 9: Response delay for each application

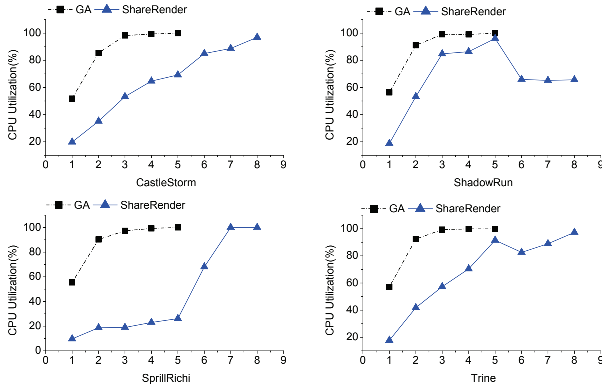
response delay for both GA and ShareRender are made up by multiple complex process. The response delay for GA and ShareRender can be calculated by is $T_{RD}^{GA} = T_{SP} + T_{CP}$ and $T_{RD}^{ShareRender} = T_{PS} + T_{PO} + T_{PE} + T_{CP}$, respectively.

We limit the network bandwidth between server and client to simulate real-world network connection. We consider three possible levels of outbound bandwidth: unlimited (100Mbps), 9.8Mbps, and 5Mbps as LiveRender [18] suggests. We also insert testing codes into GA and ShareRender to obtain the delay of different sub-processes. We manually trigger at least 100 user inputs for each game and measure the average response delay. The experiments are conducted under local area network and the physical network delay is negligible, so as to the network delay inside date center.

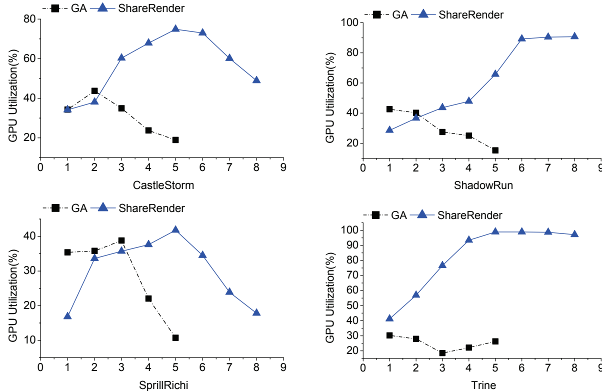
Fig. 9 shows the response delay of different games under three bandwidth conditions. We use default encoding parameters given by GA. ShareRender outperforms GA in all cases. For example, as shown in Fig. 9(d) with unlimited bandwidth, response delay of GA is 241.2ms, while it is 202.8 ms in ShareRender, with 18.93% improvements. The main contributor is faster frame capturing in ShareRender.

5.3 Concurrency

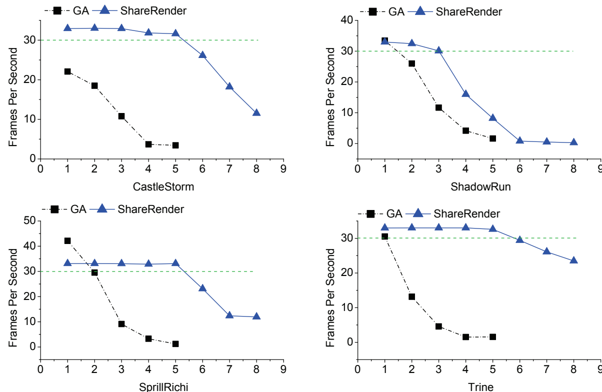
An important metric of cloud gaming systems is the number of concurrent game instances in one physical machine. We run different number of game instances under ShareRender and GA, respectively, and measure their FPS, CPU/GPU utilization. As shown in Fig. 10, FPS drops sharply under GA but remains steadily under ShareRender as the number of concurrent game instances increases. GA can support smooth playing of one or two game instances on the server, while ShareRender is able to run more. Moreover, the resource utilization of ShareRender is also improved. ShareRender is integrated with two kinds of video encoding hence it can take use of free computing resources in video encoding. Note that, the GPU utilization decreases for CastleStorm and SprillRichi in Fig. 10(b), the reason is that, after running 5 instances, ShareRender uses CPU to encode videos for new instances, which results in the longer waiting time for GPU and the decrease of GPU utilization. It is also can be seen that the FPS drops fast when both resource are exhausted



(a) CPU usage under different games



(b) GPU usage under different games



(c) FPS of different games

Figure 10: Concurrency for each application

for ShareRender while the FPS for GA drops earlier than ShareRender when CPU is exhausted.

5.4 System Overhead

We first study the overhead of memory occupation by ShareRender. We list the number of tracked objects and associated memory size under different games in Table 1. The number of shadow objects varies from hundreds to thousands, but their total size is less than 220MB under all games, which is only 14.3% of memory occupied by VMs. We notice that texture data consumes most of the memory and the texture data is static once it is loaded, hence most of the overhead in agent migration lies in initializing the texture data in context. It is one of reason for designing multiple render agents and assigning graphic tasks to existing agents.

Table 1: Traced objects and memory consumed by three types of geometry objects

Game	Total	Vertex buffer	Index buffer	Texture
CastleStorm	#17129/ 219.45 MB	#6172/ 97.15 MB	#5174/ 11.03 MB	#522/ 110.99 MB
ShadowRun	#1239/ 141.17 MB	#329/ 3.65 MB	#328/ 0.733	#302/ 136.69 MB
SprillRichi	#445/ 155.85 MB	#10/ 10.22 MB	#0/0 MB	#411/ 145.63 MB
Trine	#4300/ 197.50 MB	#1607/ 85.57 MB	#1667/ 6.24 MB	#715/ 105.40 MB

5.4.1 Render Agent Migration . We measure the migration delay as the time between issuing migration commands and obtaining the first frame after migration. We compare the agent migration and VM migration under different games, and show the results in Table 2. The agent migration in ShareRender can be done in less than 3 seconds, almost 15x faster than VM migration. That is because VM migration needs to copy the whole VM memory to the destination, but render agent migration involves only context synchronization with less data transmission. We also show the number of recreated and updated objects and its portion during agent migration in the fourth column of Table 2. We observe that at most 13.03% of the geometric objects are transmitted to the new agent during agent migration.

Table 2: Migration delay of two schemes

Game	Agent Migration	VM Migration	#Created & Updated
CastleStorm	1.46 s	43.77 s	821 / 4.79%
ShadowRun	2.74 s	41.82 s	118 / 9.53%
SprillRichi	2.06 s	30.85 s	58 / 13.03%
Trine	1.77 s	53.54 s	402 / 9.35%

5.4.2 Impact of Shadow Object. We then study the performance improvement of shadow object in ShareRender. We measure the frame processing delay of different games by enabling and disabling shadow objects, respectively. We normalize the results to the frame processing delay of disabling shadow objects. As shown in Fig. 11, enabling shadow object in ShareRender significantly reduces the frame processing time by 37.99% in average. Shadow objects are especially effective in games that update many vertices and create new textures in frames. SprillRichi is one of such games and shadow objects can reduce its frame processing delay by 80%.

5.5 Large-scale Simulation

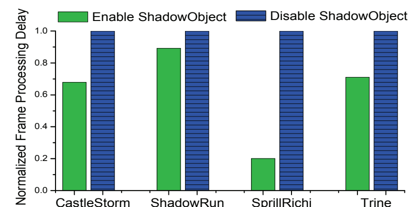
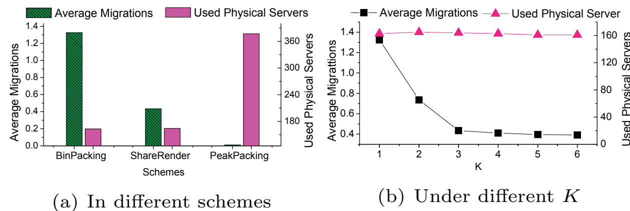


Figure 11: Normalized frame time



(a) In different schemes (b) Under different K

Figure 12: Used physical servers and average agent migrations under different K

We conduct large-scale simulations using real game traces to evaluate the proposed scheduling algorithm. The traces are generated when multiple players play the games. We consider two baselines. One is typical bin packing algorithm that minimizes the number of active servers without considering migration cost. The other one is called peak packing, which uses peak resource demands for each game instance. Hence, it requires the most physical servers but without agent migration. As shown in Fig. 12(a), traditional bin packing requires the least physical servers, while peak packing requires the most. Our algorithm achieves a good tradeoff between requirement of physical servers and agent migration overhead. Specifically, our algorithm requires only 1.006x physical servers compared to traditional bin packing, but it reduces 67.3% of the agent migrations. Peak packing uses 0.00971 agent migrations on average but requires 2.29x physical server compared to ours.

We also study the influence of number of render agents (denoted by K) and show the results in Fig. 12(b). The number of required servers slightly changes as K grows, while the number of agent migrations reduces greatly. The reason lies in that, with larger K, more candidate render agents are available for graphic tasks, leading to smaller probability of agent migration. However, when $K > 3$, the decrease of agent migration is too small. Note that, more render agents consume more additional memory and CPU/GPU resources, hence we select $K = 3$ in ShareRender.

6 RELATED WORKS

The state-of-art cloud gaming systems mainly focus on system performance optimization. GamingAnyWhere (GA) [13] is the first open-source cloud gaming system based on video streaming, which provides mature design in frame data capturing, video encoding, and input handling. However, GA lacks consideration of the concurrency and scheduling in cloud. Game@Large [14, 15] project is the first research project that majors in graphic streaming which provides the early concept of graphic streaming and 3D interception. The main limitation is high bandwidth consumption. LiveRender [18] has been proposed to reduce bandwidth consumption by applying a set of compression techniques. However, the requirement for clients limits its portability. However, it illustrated the design and implementation of graphics workloads offloading. Rhizome [22] focuses on deploying cloud gaming on latest hardware environment and illustrates the utilization of hardware in cloud gaming. [25, 30] introduce a real-time video encoding method with 3D image warping assistance. CloudFog [19] uses supernodes for game rendering and streaming to users nearby, as a result, CloudFog reduces the traffic, latency as well as bandwidth consumption.

There are works re-thinking the architectures of game engine in cloud gaming, such as [20]. Developers will be more free in designing a cloud gaming based application, but this approach sacrifices some generality compared to existing cloud gaming architectures, which are transparent to game engines and developers, such as [13, 18] and ShareRender.

Some works in cloud gaming focus on scheduling algorihms to optimize service cost. [9] considers multiplayer cloud gaming and formulates an server allocation problem to minimize the total required servers and bandwidth cost. [17] focuses on how to dispatch the play request to cloud servers in cloud gaming systems for efficient virtual machine usage. There are some other works focus on improving service quality in mobile cloud gaming, such as [16, 27].

Resource scheduling in cloud attracts many research attentions. VGRIS [31]/vGASA [32] is a resource isolation system for cloud gaming. It schedules the render pipeline on hypervisor to isolate GPU use of multiple virtual machines. GCloud [34] uses user level virtualization technique to schedule cloud gaming problems and abstracts the scheduling as bin-packing, however the idea remains scheduling the whole task and lacks strategy of solving overload. [10] maximizes gamers' experience by optimally adapting cloud gaming sessions in dynamic environments. The successor work [11] of GA studies the placement of VMs to optimize cloud gaming experience, which is still a coarse-grained scheduling far from optimal. [28] adopts dynamic resource allocation using virtualization technology. However their strategies are more suitable for latency insensitive tasks. [33] presents the fine-grained scheduling concept in cloud datacenters. They exploit the division between short and long jobs and use a constraint programming solution to schedule long jobs while uses simple heuristics to schedule short jobs. Tetris [21] is a multi-resource scheduler that adapts heuristics for the multidimensional bin packing problem. Both of [33] and [21] consider the job duration in their scheduling, but ShareRender explores more fine-grained scheduling.

7 CONCLUSION

In this paper, we present the ShareRender, a cloud gaming system bypasses GPU virtualization and enables fine-grained resource sharing in cloud gaming. ShareRender offloads graphic workloads within VMs to GPUs with the novel design of graphic wrappers and render agents. Moreover, we propose an online algorithm to determine the workloads assignment and render agent migration, which considers the tradeoff between minimizing the number of active servers and low agent migration cost. Finally, both experiments and simulations are conducted to show ShareRender outperform existing cloud gaming systems based on video streaming.

8 ACKNOWLEDGMENTS

This research is supported by National 863 Hi-Tech Research and Development Program (No.2015AA01A203), the International Science and Technology Cooperation Program of China (No. 2015DFE12860).

REFERENCES

- [1] 2016. IDirect3DDevice9::GetFrontBufferData method. https://msdn.microsoft.com/en-us/library/bb174388(v=vs.85).aspx. (2016).
- [2] 2016. Live555. http://www.live555.com/. (2016).
- [3] 2016. SDL (Simple DirectMedia Layer). http://www.libsdl.org/. (2016).
- [4] 2016. Why is GPU-CPU transfer slow? https://www.opengl.org/discussion_boards/showthread.php/168710-Why-is-GPU-CPU-transfer-slow. (2016).
- [5] 2017. Accelerate Virtual Desktops in Data Center — NVIDIA GRID — NVIDA. http://www.nvidia.com/object/nvidia-grid.html. (2017).
- [6] 2017. RFC 2326 - Real Time Streaming Protocol (RTSP). (2017).
- [7] 2017. Texture Filtering with Mipmaps (Direct3D 9). https://msdn.microsoft.com/en-us/library/windows/desktop/bb206251(v=vs.85).aspx/. (2017).
- [8] W. Cai, R. Shea, C. Y. Huang, K. T. Chen, J. Liu, V. C. M. Leung, and C. H. Hsu. 2016. The Future of Cloud Gaming [Point of View]. *Proc. IEEE* 104, 4 (2016), 687–691.
- [9] Y. Deng, Y. Li, X. Tang, and W. Cai. 2016. Server Allocation for Multiplayer Cloud Gaming. In *Proceedings of the ACM on Multimedia Conference (MM'16)*. 918–927.
- [10] H. Hong, C. Hsu, T. Tsai, C. Huang, K. Chen, and C. Hsu. 2015. Enabling adaptive cloud gaming in an open-source cloud gaming platform. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 2078–2091.
- [11] H. J. Hong, D. Y. Chen, C. Y. Huang, K. T. Chen, and C. H. Hsu. 2015. Placing Virtual Machines to Optimize Cloud Gaming Experience. *IEEE Transactions on Cloud Computing* 3, 1 (2015), 42–53.
- [12] H. Hsu and C. Lee. 2016. G-KVM: A Full GPU Virtualization on KVM. In *Proceedings of IEEE International Conference on Computer and Information Technology (CIT'16)*. 545–552.
- [13] C. Huang, C. Hsu, Y. Chang, and K. Chen. 2013. GamingAnywhere: an open cloud gaming system. In *Proceedings of the ACM multimedia systems conference (MMSys'13)*. ACM, 36–47.
- [14] A Jurgelionis, F Bellotti, AD Gloria, P Eisert, JP Laulajainen, and A Shani. 2009. Distributed video game streaming system for pervasive gaming. *STreaming Day* 9 (2009), 1–6.
- [15] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. P. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perälä, A. De Gloria, and C. Bouras. 2009. Platform for Distributed 3D Gaming. *Int. J. Comput. Games Technol.* 2009, Article 1 (2009), 15 pages.
- [16] K. Lee, D. Chu, C. Eduardo, K. Johannes, D. Yury, Sergey G., W. Alec, and F. Jason. 2015. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'15)*. ACM, 151–165.
- [17] Y. Li, X. Tang, and W. Cai. 2015. Play Request Dispatching for Efficient Virtual Machine Usage in Cloud Gaming. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 2052–2063.
- [18] L. Lin, X. Liao, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li. 2014. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. In *Proceedings of ACM International Conference on Multimedia (MM'14)*. 347–356.
- [19] Y. Lin and H. Shen. 2017. CloudFog: Leveraging Fog to Extend Cloud Gaming for Thin-Client MMOG with High Quality of Service. *IEEE Transactions on Parallel and Distributed Systems* 28, 2 (2017), 431–445.
- [20] F. Messaoudi, G. Simon, and A. Ksentini. 2015. Dissecting games engines: The case of Unity3D. In *Proceedings of Annual Workshop on Network and Systems Support for Games (NetGames'15)*. 1–6.
- [21] G. Robert, A. Ganesh, K. Srikanth, R. Sriram, and A. Aditya. 2014. Multi-resource Packing for Cluster Schedulers. In *Proceedings of ACM International Conference on SIGCOMM (SIGCOMM'14)*. 455–466.
- [22] S. Ryan, D. Fu, and J. Liu. 2015. Rhizome: Utilizing the Public Cloud to Provide 3D Gaming Infrastructure. In *Proceedings of ACM Multimedia Systems Conference (MMSys'15)*. 97–100.
- [23] S. Ryan and J. Liu. 2013. On GPU Pass-Through Performance for Cloud Gaming: Experiments and Analysis. In *Proceedings of Annual Workshop on Network and Systems Support for Games (NetGames'13)*. 6:1–6:6.
- [24] R. Shea, J. Liu, E. C. H. Ngai, and Y. Cui. 2013. Cloud gaming: architecture and performance. *IEEE Network* 27, 4 (2013), 16–21.
- [25] S. Shu, N. Klara, and C. Roy. 2012. A Real-time Remote Rendering System for Interactive Mobile Graphics. *ACM Trans. Multimedia Comput. Commun. Appl.* 8, 3s, Article 46 (Oct. 2012), 20 pages.
- [26] K. Tian, Y. Dong, and D. Cowperthwaite. 2014. A Full GPU Virtualization Solution with Mediated Pass-Through. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC'14)*. 121–132.
- [27] J. Wu, C. Yuen, N. Cheung, J. Chen, and C. Chen. 2015. Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 1988–2001.
- [28] Z. Xiao, W. Song, and Q. Chen. 2013. Dynamic Resource Allocation Using Virtual Machines for Cloud Computing Environment. *IEEE Transactions on Parallel and Distributed Systems* 24, 6 (2013), 1107–1117.
- [29] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan. 2016. gScale: scaling up GPU virtualization with dynamic sharing of graphics memory space. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC'16)*. 579–590.
- [30] W. Yoo, S. Shi, W. J. Jeon, K. Nahrstedt, and R. H. Campbell. 2010. Real-time parallel remote rendering for mobile devices using graphics processing units. In *Proceedings of IEEE International Conference on Multimedia and Expo (ICME'10)*. 902–907.
- [31] M. Yu, C. Zhang, Z. Qi, J. Yao, Y. Wang, and H. Guan. 2013. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *Proceedings of International Symposium on High-performance Parallel and Distributed Computing (HPDC'13)*. 203–214.
- [32] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan. 2014. vGASA: Adaptive Scheduling Algorithm of Virtualized GPU Resource in Cloud Gaming. *IEEE Transactions on Parallel and Distributed Systems* 25, 11 (2014), 3036–3045.
- [33] Y. Zhang, X. Fu, and K. K. Ramakrishnan. 2014. Fine-grained multi-resource scheduling in cloud datacenters. In *Proceedings of International Workshop on Local Metropolitan Area Networks (LANMAN'14)*. 1–6.
- [34] Y. Zhang, P. Qu, J. Cihang, and W. Zheng. 2016. A Cloud Gaming System Based on User-level Virtualization and Its Resource Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1239–1252.