gRemote: API-Forwarding Powered Cloud Rendering

Dongjie Tang, Yun Wang, Linsheng Li
Shanghai Jiao Tong University
Shanghai, China
{018033210001,yunwang94,lilinsheng1}@sjtu.edu.cn

Jiacheng Ma
University of Michigan
Michigan, USA
jcma@umich.edu

Xue Liu
McGill University
Montreal, Canada
xueliu@cs.mcgill.ca

Zhengwei Qi, Haibing Guan
Shanghai Jiao Tong University
Shanghai, China
{qizhwei,hbguan}@sjtu.edu.cn

ABSTRACT

Traditional GPU resource allocation approaches, widely adopted in today's data centers, only focus on the server-side functions while ignoring the client-side. These approaches waste client-side hardware resources. To solve this problem, remote API-forwarding architectures appear. Through running applications on the client-side, remote API-forwarding architectures offload some workloads to the client. However, many remote API-forwarding systems suffer from one big issue: shared-resource interference, stemming from two reasons: (a) GPU resource racing caused by resource overuse for a single client, and (b) CPU resource racing caused by resource shortage among clients. This paper presents gRemote, an open-source GPU-remoting system that can address this issue. To mitigate the CPU resource shortage, gRemote improves CPU configurations by expanding CPU resources from the server-side to both server- and client-side. To maintain the reasonable GPU usage for individual tasks, we innovate a new resource-sharing mechanism called *GPU throttle*. gRemote supports 1,228 OpenGL commands with around 10% shared-resource interference.

KEYWORDS

gRemote, Cloud Rendering, API-Forwarding, OpenGL

ACM Reference Format:
Dongjie Tang, Yun Wang, Linsheng Li, Jiacheng Ma, Xue Liu, and Zhengwei Qi, Haibing Guan. 2020. gRemote: API-Forwarding Powered Cloud Rendering. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '20), June 23–26, 2020, Stockholm, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3369583.3392676

1 INTRODUCTION

With the rapid development of computing resources in personal devices, leaving all workloads to servers is not the only choice. Unlike the thin-client architecture[1] pushing the entire application to the server, the *API-forwarding* architecture only pushes acceleration tasks to the cloud. Thus, the remote API-forwarding system has the potential to fully utilize the client-side resources and offers GPU acceleration for both *intra-cloud* and *cloud-edge* scenarios, making itself an alternative of the thin-client architecture. In this paper, we focus on optimizing this architecture.

Previous work on API-forwarding systems either focuses on GPGPU computation (e.g., rCUDA [1]), or optimization of command and data transmission (e.g., LiveRender [6]). Thus, the problem of *shared-resource interference*, which hurts the performance of API-forwarding powered cloud rendering systems, remains unsolved. Shared-resource interference is a scenario that applications race for resources on the server. It stems from one of the two aspects: (a) GPU resource racing, and (b) CPU resource racing. *GPU resource racing* happens when the GPU resource is rich enough; with an ineffective sharing method, one of the applications sharing the same GPU may overuse the whole GPU. This behavior may hurt the performance of other applications. *CPU resource racing* happens when the CPU resource becomes fully-utilized; in this scenario, applications on the server race for resources to perform CPU-related tasks (e.g., transferring commands to standard GPU APIs and compressing frames). Although there are mechanisms [5, 7] seeking to solve the resource-sharing problem by putting different applications onto one server, none of them works on remote API-forwarding systems.

This paper presents gRemote, the first open-source remote API-forwarding system for cloud rendering that can mitigate shared-resource interference. To containerize each application within a reasonable GPU usage, gRemote proposes *GPU throttle*, a technique that helps to share a GPU resource evenly among different rendering workloads. To alleviate CPU resource racing, gRemote reduces the server-side CPU resource requirement by forwarding only GPU-related APIs rather than all graphics commands, leaving as many CPU tasks done in the client-side as possible. Through network transmission, clients and servers work together to complete each task. We call this behavior *cloud-edge cooperation*.

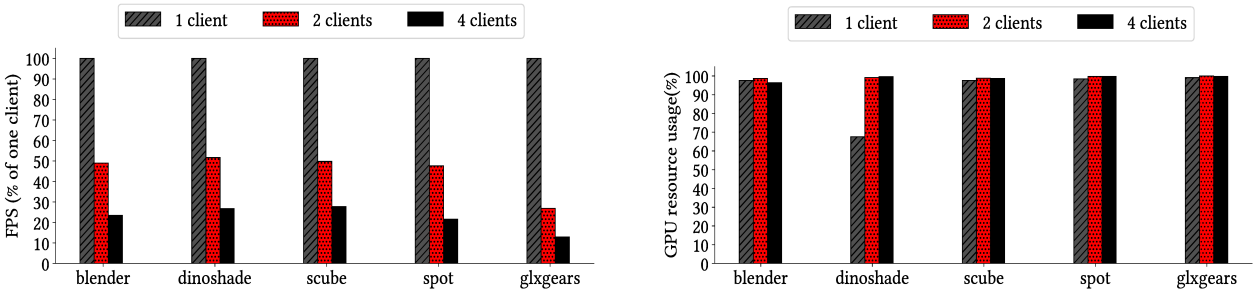
2 MOTIVATION

In this section, we describe the motivation of gRemote and demonstrate the problem of shared-resource interference.

[1] Xbox Play Anywhere. https://www.xbox.com/en-SG/games/xbox-play-anywhere

HPDC '20, June 23–26, 2020, Stockholm, Sweden

ACM ISBN 978-1-4503-7052-3/20/06…$15.00
https://doi.org/10.1145/3369583.3392676



(a) Performance variation for different applications

(b) GPU resource ratio taken by different applications

Figure 1: Performance degradation under the intra-cloud scenario

2.1 Inefficiency of GPU Resource Sharing

We conducted the experiments on two machines: one configured with Nvidia Geforce 1050 Ti as the server and the other without any dedicated graphics card as the client. Both machines are configured with RDMA. As shown in Fig. 1 (a), we choose four applications: glxgears widely used in previous research [2, 3], and three microbenchmarks from the OpenGL official site[2]. We take FPS, a well-recognized metric [7] to measure the performance of graphics applications and nvidia-smi [3] to measure the GPU usage. Due to different FPS values of different benchmarks, we take the FPS of every single application as 100% and normalize the remaining values based on this FPS for the same application. In this experiment, we make sure hardware resources are rich enough to handle these tasks.

Despite rich hardware resources, performance degradation happens when the number of clients increases. The biggest drop happens when client number increases from 1 to 2, reaching more than 60%. *Glxgears* even has more than 80% loss. Thus, performance degradation appears even if hardware resources are rich enough.

We monitor the hardware status when running four applications. As shown in Fig. 1 (b), even if different applications have different GPU resource requirements to meet its QoS (around 30 FPS ∼ 60 FPS), GPU resources are almost fully-utilized, leading to an unreasonable high FPS (e.g., blender reaches more than 10000 FPS). A new application tends to steal resources from the original one and causes great performance loss.

Based on these experiments, we conclude that without an effective sharing mechanism, each application tends to overuse the entire GPU resource and causes performance degradation when a new application arrives. This kind of shared-resource interference happens when hardware resources are enough.

2.2 CPU Resource Bottleneck

Another type of shared-resource interference comes when the CPU resource is fully utilized.

Algorithm 1 shows the execution steps of applications, except for *PipelineCreated* and *GPURendering*, the remaining tasks belong to the CPU. Furthermore, for some special applications including human-interactive ones, *IOProcessing* is required periodically, which means that the CPU needs to handle more workloads.

```
System_call_IOProcessing();
TransCommands_to_GPU();
Context_establish_Initialization();
while true do
    PipelineCreated_FrameSetup();
    GPURendering_BufferReady();
    Present();
end
```

Algorithm 1: Workloads of Rendering Applications

Based on the workload distributions, all rendering tasks require more CPU than GPU. However, current remote API-forwarding systems tend to forward all rendering commands to the cloud. Many of them are done by CPU including context creation and command translations. Thus, the server-side CPU is still overwhelmed. Many remote API-forwarding systems intensify the CPU bottleneck issue by allowing multiple applications to share on one server.

Thus, gRemote goes one further step to improve CPU resources for each application. At the same time, gRemote innovates its mechanism to avoid GPU racing happen.

3 ARCHITECTURE

In this section, we present the whole architecture of gRemote and discuss how gRemote lowers resource-sharing interference from two aspects: 1) alleviating the server-side CPU bottleneck; 2) containerizing each application within a reasonable GPU usage. As shown in Fig. 2, gRemote can be divided into two parts: *computing part* (client-side) and *rendering part* (server-side). The computing side includes graphics context establishment and commands translations. Rendering side incorporates 2D or 3D graphics rendering and pipeline creation. On the computing part, gRemote leaves as many CPU workloads as possible to mitigate CPU racing caused by CPU shortage. On the rendering part, gRemote uses *GPU throttle*, a resource-sharing mechanism, to minimize GPU racing caused by GPU resource overuse.

[2]https://www.khronos.org/opengl/wiki/Code_Resources
[3]https://developer.nvidia.com/nvidia-system-management-interface

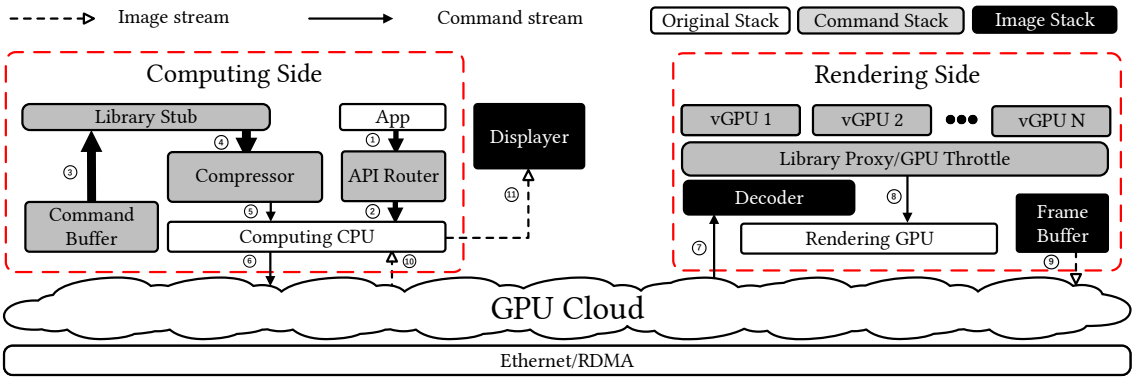


Figure 2: The detailed architecture of gRemote

3.1 Client-Side Architecture

Instead of exhausting every effort to save tiny server-side CPU resources, gRemote offloads CPU-computing tasks to the client-side. Through cooperation between the cloud and the edge, gRemote greatly mitigates the server-side CPU pressure.

API Router: gRemote provides an *API router*, responsible for dividing applications into two parts: the computing part and the rendering part. Through the *API router*, these two parts are dispatched to different places. For instance, computing workloads leave on the client-side; rendering workloads are remoted to the server-side. Instead of sending all rendering commands to the server-side immediately, gRemote leaves them on the local-side.

Command Buffer: Instead of transmitting commands one by one, gRemote establishes a *command buffer* as a transmitting unit. gRemote stores all the prepared commands into the buffer. Then, gRemote sends it to the *library stub*.

Library Stub: In gRemote, *library stub* divides applications further into the GPU-related part and the CPU-related part. Even though the *API router* already did application divisions, there are still many CPU workloads left (e.g., context establishment, window setting, and I/O processing). Thus, we leave these to the *computing side* locally and send the rest to servers after compression.

However, gRemote cannot eliminate CPU workloads on the server-side because applications still need CPU to drive GPU and GPU still needs CPU to process standard OpenGL commands to GPU-recognized ones. Besides minimizing server-side CPU requirements, such a behavior saves the server-side execution time, paving a way for more clients using one hardware resource.

3.2 Server-Side Architecture

Besides the CPU resource bottleneck, another reason causing shared-resource interference comes from GPU resource overuse. We will analyze why GPU resource overuse appears and how *GPU Throttle* solves it.

For many rendering applications, execution time completely depends on user behaviors. However, without effective methods, the speed mismatch between frame creation and rendering [4] makes GPU use more cores to cover. Thus, any application can take almost all the GPU resources and force new applications to steal resources. Such a stealing behavior causes original applications to lose resources without any notification, leading to serious performance degradation.

```
Input: [Qlow , Qhigh], sched_interval
sync frame of client_i;
T = CalcTimeSpent();
if T ⩾ sched_interval then
    FPS[client_i] = GetFPS(client_i);
    while FPS[client_i] not belong to [Qlow, Qhigh] do
        Readjust frame creation speed;
        if failed allocation then
            FPS[client_i] =0;
            Response_value[client_i]=0;
        end
    end
end
Yield();
```

Algorithm 2: GPU Throttle

Based on the GPU characteristics, we propose our mechanism: *GPU throttle*, dynamically containerizing resources based on current resource usage of each client and its QoS. As shown in Algorithm 2, *GPU throttle* uses FPS as a signal to control the GPU usage of each client. Since it is hard to regulate the speed of one application to a certain value, we establish a speed range (*Qlow* ~ *Qhigh*). *Qlow* means the QoS of each application and *Qhigh* means the 1.2*QoS of each application. As long as FPS falls into that range, *GPU throttle* sends a success signal. Otherwise, it fails. Based on the analysis before, when FPS does not fall into the speed range, *GPU throttle* adjusts the speed of frame creation until it succeeds. There is one failing case that the application fails to reach its *Qlow* without any speed reduction. In that case, *GPU throttle* will not spare any resource to that application and set FPS to 0.

GPU Throttle isolates different instances using vGPUs of various sizes, independent from hardware and operating systems. From the server-side, each vGPU represents a client's requirement.

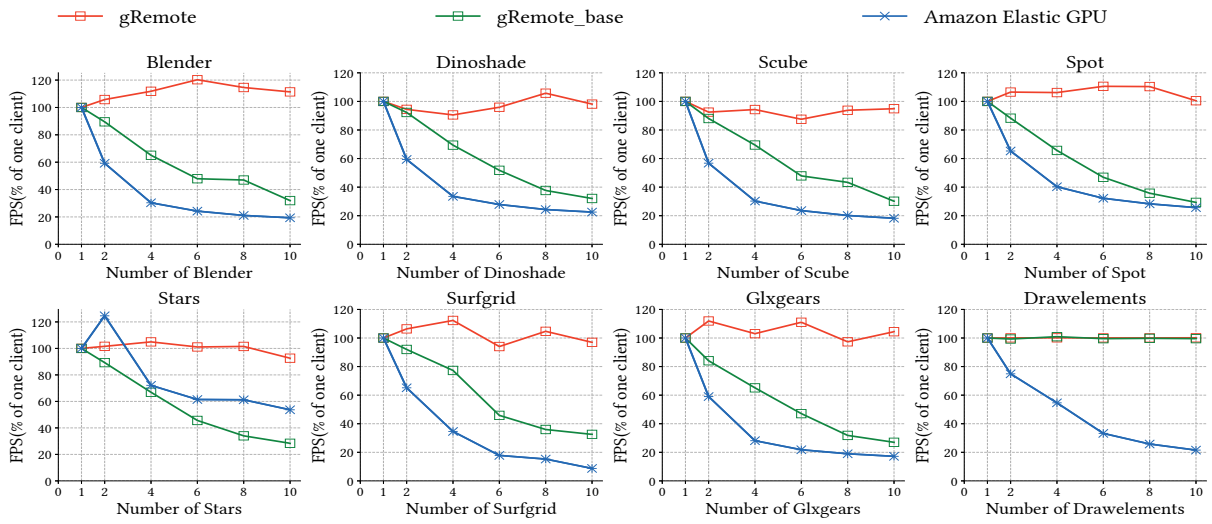


Figure 3: Performance interference of multi-clients among *gRemote*, *gRemote_base*, and *Amazon Elastic GPU*

4 EVALUATION

In this section, we evaluate *gRemote* to see the shared-resource interference and hardware usage of one application.

4.1 Experimental Setup

Hardware configuration: *gRemote* has two components: a gRemote server (for rendering and streaming videos), a gRemote client (for sending graphic commands and showing results). We establish a gRemote server with one NVIDIA 1060 and Intel Core i5. We configure a gRemote client only with Intel Core i5.

Baseline: To make sure the fairness and comprehensiveness, we take two baselines: *gRemote_base*, a traditional remote API-forwarding system(i.e., without GPU throttle and CPU resource improvement) and *Amazon Elastic GPU* (a closed-source and state-of-art commercial product which has similar targets with *gRemote*).

Benchmarks: We use 8 benchmarks to evaluate *gRemote*, *gRemote_base*, and *Amazon Elastic GPU*: 6 micro-benchmarks obtained from OpenGL official site and 2 famous benchmarks, *glxgears* and *drawelements*.

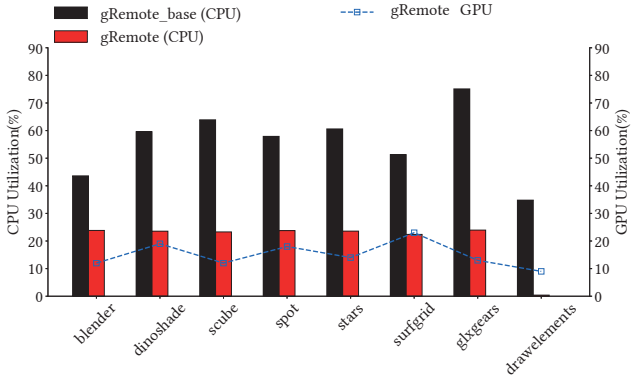


Figure 4: The optimization of hardware utilization

4.2 Shared-resource Interference

To evaluate performance interference in *gRemote*, *Amazon Elastic GPU*, and *gRemote_base*, we take FPS as a metric. However, we do not directly use real FPS values because *Amazon Elastic GPU* is closed-source and no technical information can be obtained from its server-side. Thus, we take FPS of every single application as 100% and normalize the remaining values based on this FPS for the same application. Fig. 3 presents the comparing results. For *gRemote*, the performance variation from the different numbers of applications is within 10%. For *Amazon Elastic GPU* and *gRemote_base*, when clients number exceeds 2, the performance loss appears. There are two exceptions in Fig. 3. For *Stars* applications in the Amazon part, FPS of a single one is worse than that of two applications because data caching speeds up the whole execution [4]. For *drawelements*, it has its mechanisms to do resource control. Thus, *gRemote* doesn't do anything for this benchmark.

Fig. 4 presents the hardware resource usage of a single application in *gRemote* and *gRemote_base*.

Because no server information can be obtained from *Amazon Elastic GPU*, we only represent the hardware usage of *gRemote* and *gRemote_base*. On the CPU part, we can see the optimization brought by *gRemote*. In the *gRemote_base*, all the applications, both micro and macro ones, tend to occupy more than 50% of CPU. Through offloading the server-side CPU workloads to the client-side, *gRemote* lowers the CPU usage by more than 60%. On the GPU part, instead of maximizing GPU resources for each application, *GPU throttle* containerizes each application within a reasonable GPU part to meet the QoS of each application.

ACKNOWLEDGEMENT

Thanks for the help from our shepherd: Douglas Thain. This work was supported in part by National Key Research & Development Program of China (No.2016YFB1000502), National NSF of China (NO. 61672344, 61525204, 61732010), and Shanghai Key Laboratory of Scalable Computing and Systems.

REFERENCES

[1] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proceedings of the 2010 International Conference on High Performance Computing & Simulation, HPCS 2010, June 28 - July 2, 2010, Caen, France*, Waleed W. Smari and John P. McIntire (Eds.). IEEE, 224–231. https://doi.org/10.1109/HPCS.2010.5547126

[2] Cheol-Ho Hong, Ivor T. A. Spence, and Dimitrios S. Nikolopoulos. 2017. FairGV: Fair and Fast GPU Virtualization. *IEEE Trans. Parallel Distrib. Syst.* 28, 12 (2017), 3472–3485. https://doi.org/10.1109/TPDS.2017.2717908

[3] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. 2002. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.* 21, 3 (2002), 693–702. https://doi.org/10.1145/566654.566639

[4] John Kessenich, Graham Sellers, and Dave Shreiner. 2016. *OpenGL Programming Guide: The Official Guide to Learning Opengl, Version 4.5 with Spir-V.* Addison-Wesley.

[5] Yusen Li, Chuxu Shan, Ruobing Chen, Xueyan Tang, Wentong Cai, Shanjiang Tang, Xiaoguang Liu, Gang Wang, Xiaoli Gong, and Ying Zhang. 2019. GAugur: Quantifying Performance Interference of Colocated Games for Improving Resource Utilization in Cloud Gaming. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing - HPDC '19*. 231–242.

[6] Li Lin, Xiaofei Liao, Guang Tan, Hai Jin, Xiaobin Yang, Wei Zhang, and Bo Li. 2014. LiveRender: A Cloud Gaming System Based on Compressed Graphics Streaming. In *Proceedings of the 22nd ACM international conference on Multimedia*. 347–356.

[7] Miao Yu, Chao Zhang, Zhengwei Qi, Jianguo Yao, Yin Wang, and Haibing Guan. 2013. VGRIS: virtualized GPU resource isolation and scheduling in cloud gaming. In *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*, Manish Parashar, Jon B. Weissman, Dick H. J. Epema, and Renato J. O. Figueiredo (Eds.). ACM, 203–214. https://dl.acm.org/citation.cfm?id=2462914