PilotFish: Harvesting Free Cycles of Cloud Gaming with Deep Learning Training

Wei Zhang*
Shanghai Jiao Tong University

Binghao Chen
Shanghai Jiao Tong University

Zhenhua Han
Microsoft Research Asia

Quan Chen
Shanghai Jiao Tong University

Peng Cheng
Microsoft Research Asia

Fan Yang
Microsoft Research Asia

Ran Shu
Microsoft Research Asia

Yuqing Yang
Microsoft Research Asia

Minyi Guo
Shanghai Jiao Tong University

Abstract

Cloud gaming services have become important workloads in cloud datacenter. However, our investigation shows that a cloud gaming service cannot saturate the modern cloud GPUs. One way to improve the GPU utilization is to co-locate multiple workloads within one GPU, which is challenging for cloud gaming due to its highly fluctuated and unpredictable GPU usage pattern. In this paper, we present PilotFish, a high-performance system that harvests the free GPU cycles of cloud gaming with deep learning (DL) training, while incurring almost zero interference to cloud gaming. We co-locate DL training jobs with cloud gaming, because they have stable and predictable workloads and have no strict latency requirement. In more detail, PilotFish captures the idle periods of the game's GPU usage with its low-overhead instrumentation to graphic libraries in sub-millisecond granularity. To avoid the potential interference to cloud gaming, PilotFish schedules training computation kernels only when they can finish before the idle GPU periods, and preempts straggler kernels running longer than expected. Our evaluation on popular cloud games and DL models shows PilotFish can harvest up to 85.1% of the idle GPU time from cloud gaming with no interference.

1 Introduction

Cloud gaming is gaining popularity in recent years. As shown in Figure 1, players of cloud gaming only use a thin client that interacts with games running on cloud servers and receives the stream of rendered frames via Internet [38]. Cloud gaming greatly reduces the hardware requirement of high-quality video games. Mobile clients with no or weak GPU can still enjoy the good visual effect of powerful GPUs. Cloud gaming has become an important workload in major cloud service providers, e.g., Microsoft's Xbox Remote Play [11], Google's Stadia [7], Nvidia's Geforce Now [15], Sony's PlayStation Now (running on Azure) [18], Amazon's AppStream [1].

*This work is done while Wei Zhang is an intern in Microsoft Research.

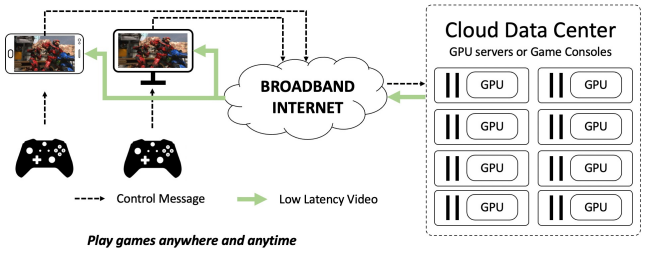


Figure 1: In cloud gaming games, players send control messages (keyboard and mouse) to cloud servers. Game scenes are rendered as frames in cloud servers and streamed to edge devices via internet.

Due to the limitations of the network, encoding and decoding capability, and resolution of mobile devices, major cloud gaming service only provides limited streaming quality that is far lower than the increasing capability of modern GPUs. For example, Microsoft's Xbox Remote Play and PlayStation Now only support up to 1080p at 60FPS. However, the latest GPUs for gaming (e.g., Nvidia's 3090Ti) can support 4K (2160p) resolution at up to 144FPS. Running cloud gaming of limited streaming quality on powerful GPUs would inevitably waste the GPU cycles. Our evaluation of popular games shows most of them have a utilization lower than 50% with cloud gaming GPUs. It is important to improve the utilization to reduce the operation cost of cloud gaming services.

To improve GPU utilization for cloud gaming, a natural solution is to co-locate multiple workloads in one GPU (e.g., multiple games [29, 36] or other GPU workloads [23–25, 50, 51]). Such approaches face great challenges, due to the high randomness of the gaming workload. A game's utilization of different resources (including GPU, CPU, PCI-e and disk I/O) varies greatly across video frames. Such variation is difficult to predict due to the random interaction between players and changing game scenes. Moreover, different games could exhibit very diverse resource usage patterns, further increasing the degree of unpredictability. Co-locating multiple games in a GPU would inevitably lead to interfer-

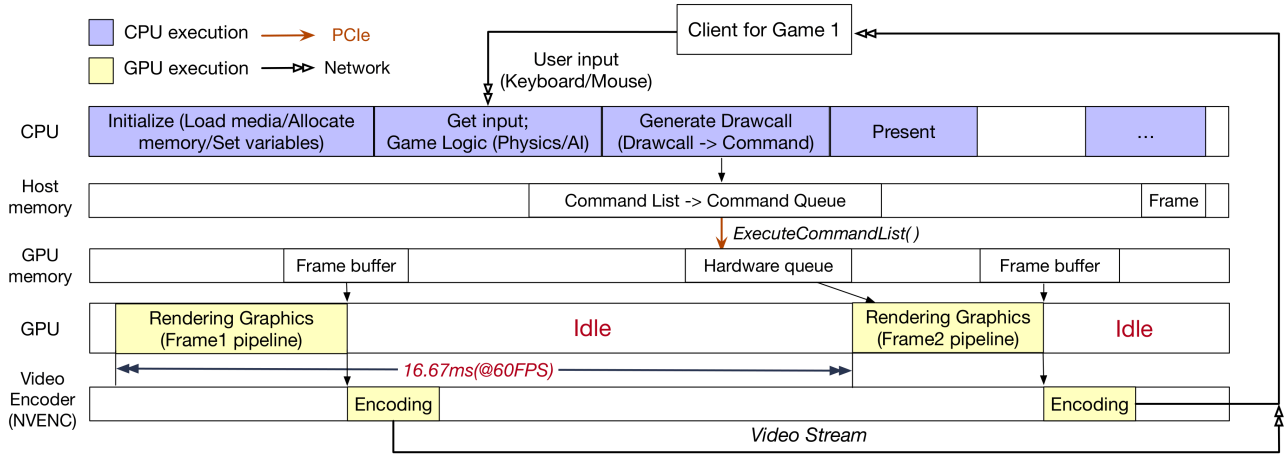


Figure 2: The procedures of cloud gaming. On receiving user input, the game logic decides the content of the scene to be rendered, which is comprised of a list of draw calls using graphic libraries. The draw calls are pushed into a command queue for the frame and submitted to the hardware for rendering the frame. The rendered frames are encoded by dedicated chips (e.g., NVENC [13] of Nvidia's GPUs) and sent to the cloud gaming client.

ence when long rendering times from different games collide. To safely harvest the GPU free cycles from cloud gaming, it is necessary to choose a more predictable and stable workload for co-location, where we find Deep Learning (DL) training, a pervasive workload in cloud data centers, is a good fit.

In this paper, we present PilotFish, a high-performance system that harvests the free GPU cycles of cloud gaming with deep learning training, without impacting gaming experiences. Instead of predicting the varying gaming workload, PilotFish exploits idle GPU periods in a reactive manner. PilotFish exposes a real-time resource monitoring interface by instrumenting graphic libraries (e.g., DirectX or Vulkan) for quickly reporting (within 10 μs) the start and completion of the rendering of a game frame. This way, PilotFish can precisely capture idle GPU periods of games. This design allows PilotFish to support all games running on common graphic libraries without modifying or re-compiling game code.

PilotFish further leverages the *predictability* of deep learning training in the scheduling. It is well-known that deep learning training consists of iterative training steps. The compute kernels in each training step have a highly predictable execution time and can be obtained through offline profiling [40,46]. With the known duration of a specific DL training kernel (usually on the order of sub-millisecond), PilotFish is able to safely schedule a deep learning training job to leverage the idle time of gaming workload, without violating the QoS of cloud gaming. The interference on other types of hardware resources is also avoided via state-of-the-art techniques, e.g., Baymax [25] for PCI-e. Furthermore, to prevent from training anomaly, where a DL training kernel does not complete in the estimated time, PilotFish can proactively terminate the training (<1ms) with limited loss of training progress.

We have implemented a prototype of PilotFish to support games on DirectX 12 [3] and DL training using Nvidia CUDA 11 [14]. We evaluate PilotFish using popular games for cloud gaming and widely-used DL models for training. Evaluation result proves PilotFish can strictly guarantee the QoS of cloud gaming when co-located with DL training. PilotFish can harvest up to 85.1% of the idle GPU time without interference, compared to straw-man baselines that degrade the 99%-ile FPS by over 30% to achieve the same harvest ratio.

The key contributions of the paper are as follows:

- We identify the low GPU utilization problem of cloud gaming and the challenges of co-location due to the randomness of games.
- We characterize the cloud gaming workload and point out that DL training is a right workload to be co-located with cloud gaming to improve GPU utilization.
- We propose mechanisms for quickly capturing idle GPU periods of gaming and fine-grained scheduling of co-located DL training workload, which guarantee no interference.

2 Motivation

In this section, we study the common cloud gaming pipeline shown in Figure 2. We investigate why there is low utilization issue in cloud gaming services and the challenges of harvesting free GPU cycles from games. Then we motivate why DL training is a good fit for co-location.

Table 1: The GPU and CPU utilization of cloud gaming.

Game	Average GPU Util.	Peak GPU Util.	VRAM (GB)	CPU Util.	FPS
Dota 2	38.2%	45%	1.61	21.9%	59.9
League of Legends	26.9%	41%	1.16	22.0%	59.8
PUBG	40.6%	95%	4.05	28.9%	60.1
CS:GO	45.0%	57%	2.6	69.7%	201
Civilization 5	32.3%	42%	1.11	15%	59.8
The Division 2	89.5%	98%	3.12	46.11%	58.66
Assassin's Creed Odyssey	69.2%	78%	2.39	66.3%	59.68
Ashes of the Singularity	89.8%	98%	3.42	79.23%	57.31

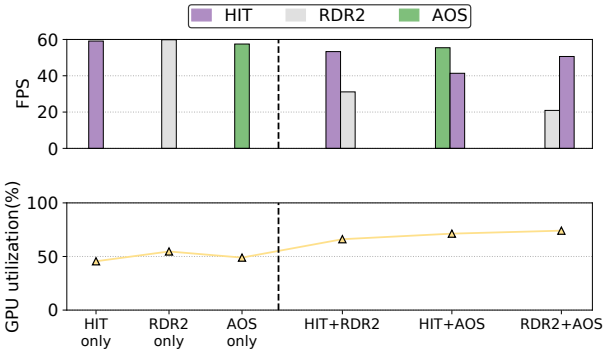


Figure 3: The average FPS and GPU utilization of independent execution of popular games and their co-located execution on Nvidia RTX 2060. (HIT: HITMAN3, RDR2: Red Dead Redemption 2 and AOS: Ashes of the Singularity)

2.1 GPU Under-utilization of Cloud Gaming

Existing cloud gaming platforms allocate each player to a dedicated server for running the requested game to ensure players' satisfactory experiences. For cloud gaming service providers, major concerns are focused on network latency and operational cost. The network latency is considerably reduced today and becomes viable for cloud gaming. However, the low resource utilization still leads to significant operation cost. We use the Nvidia RTX 2060 GPU, of which the computing ability is 6.4 teraflops, as the experimental platform, which has comparable performance to the Xbox One X's GPU (6.01 teraflops) used by Microsoft's cloud gaming service. We investigate the performance of eight of the most popular games. Table 1 summarizes the resource utilization of these games on NVIDIA RTX 2060 with cloud gaming rendering quality, mostly 1080p and 60 Frames Per Second (FPS). Five of the eight games have a GPU utilization of lower than 50%, showing the potential opportunities for improvement.

Modern GPUs are becoming more and more powerful. However, the QoS of cloud gaming is much lower than the capability of modern GPUs. According to Steam's survey [19], over 83.67% of PC gamers use resolution $\leq$ 1920x1080. Most smartphones only have a screen $\leq$ 1080p resolution. Also, the higher resolution requires better network quality and hardware capability (for decoding). Currently, Xbox remote play only supports streaming quality of at most 1080p and 60 FPS.

We anticipate the low GPU utilization issue will become more severe on the latest generation of GPUs used by gaming clouds. For example, Google Stadia uses an AMD GPU with 10.7 teraflops [7], Microsoft's Xbox Series X chip has 12 teraflops [11], Nvidia's RTX 3090 has 35.58 teraflops.

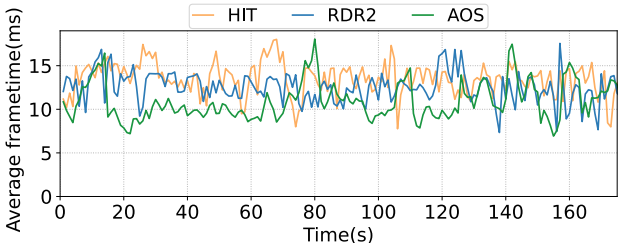


Figure 4: The fluctuation of frame time over time of three popular games.

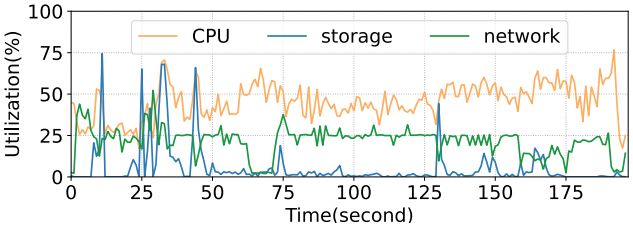


Figure 5: The fluctuation of CPU, storage and network utilization over time of Hitman 3.

2.2 Challenges

A natural idea to improve the GPU utilization of cloud gaming is to co-locate multiple games into the same GPU. However, we observe co-locating multiple games could severely interfere with each other, even when the GPU is still underutilized.

Figure 3 demonstrates the FPS of three popular cloud games and their GPU utilization in two situations: independently execution and co-located execution. When the games run alone, they can achieve around 50% of GPU utilization on 60 FPS. However, when two games are co-located, the FPS drops greatly (e.g., RDR2's FPS drops to 20 from 60) but the GPU utilization is only improved by up to 24%.

The main cause of the degraded co-location performance comes from the randomness of games. As shown in Figure 4, a game's frametime (the time to render a frame) could vary significantly over time due to the different complexity of the scene. Different games would further exhibit a very diverse pattern of GPU consumption. Moreover, in addition to GPUs, the resource usage of other resource types (CPU, storage, etc.) also fluctuates over time as illustrated in Figure 5. When contention appears on these resources, the submission of draw calls would be blocked, which also leads to lower GPU utilization. This explains why the co-located games only have

limited improvement on GPU utilization in Figure 3.

The highly random gaming behaviours make it impossible to co-locate the other random and interactive workloads like game without impacting the gaming experience. Previous works [36, 43] using static profiling to co-locate multiple games in a best-effort manner could still suffer from the interference due to random rendering content. We seek to find a more stable and predictable workload as the candidate for co-location, where we find DL training is a good fit.

2.3 Co-location with DL Training

DL training is a pervasive workload in cloud data centers. The major cloud gaming service providers (e.g., Microsoft, Google, Amazon) also have a huge demand for training DL models with GPUs [33]. The key reason we consider DL training for co-location with cloud gaming is its predictability and fine-granularity. Figure 6 shows the execution time of statistical top 20 frequent kernels from six popular DL training models. The figure shows that the duration of all training kernels is relatively stable, it usually varies within a few percent. Thus, by leveraging the predictability and iterative pattern of DL kernels, the system can know their duration *beforehand*. Also, the execution time of DL kernels is typically less than 1ms, thus it is very suitable to be scheduled to exploit the GPU idle time.

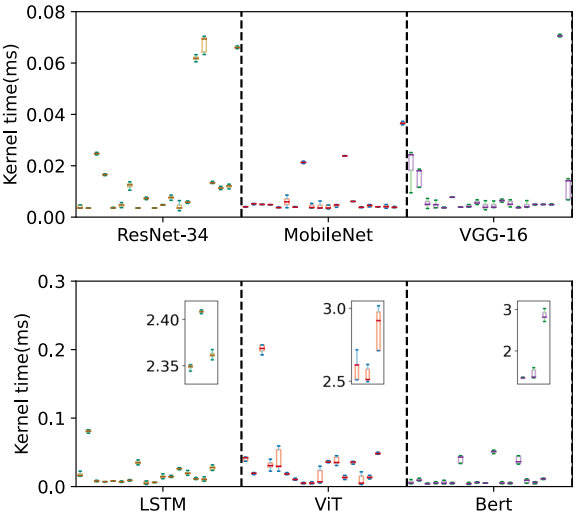


Figure 6: The execution time of top 20 frequent kernels from six popular models. (each bar is a kernel).

Despite the opportunity, the direct co-location of gaming and DL training without leveraging the characteristics of DL training could still incur a severe drop in FPS due to complex interference behaviors. For example, if the DL training kernels are submitted when a frame is still under rendering, both workloads would contend for GPU time and postpone the completion of game rendering.

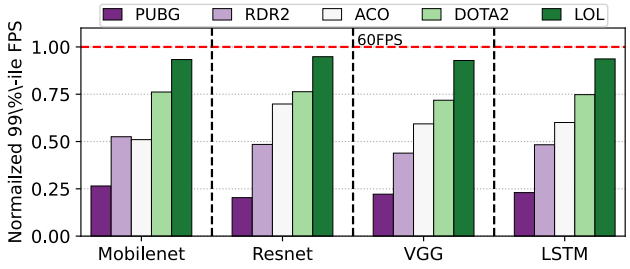


Figure 7: FPS of games when co-located with DL training.

Figure 7 shows the normalized 99%-ile FPS when five popular games are co-located with DL training tasks on a Nvidia 2060 GPU. The DL training tasks includes Resnet50 [30], VGG [44] and Mobilenet [31]. In the figure, the *x*-axis indicates the combination of games and DL training tasks, and the *y*-axis shows the 99%-ile FPS normalized to its FPS target. All games are affected by naively co-locating with DL training models. During co-location, we observe that the FPS of game is affected by the *duration of frame rendering*, the *DL training kernel scheduling* and the *contention on shared resources*. Therefore, it requires very careful management of the co-located DL training jobs to avoid interfering the cloud gaming, which is the main goal of PilotFish.

3 PilotFish Overview

We consider the scenario that DL training has a lower priority than the interactive cloud gaming service. Therefore, it is required that DL training should not generate interference to cloud gaming. PilotFish co-designs the cloud gaming services and deep learning training frameworks so that they can collaboratively work together. Figure 8 demonstrates the overall design of PilotFish.

Instead of predicting the random gaming behaviours, *PilotFish monitors the frame-level execution and resource-usage information in real time with very low overheads.* Existing frame monitoring tools [8, 9, 17] for gaming are usually based on event-tracing technology [4], which is for general-purpose application by design and infeasible for PilotFish's requirement due to its high latency. To capture the idle GPU periods, PilotFish instruments the graphic libraries to quickly and precisely detect when the rendering of a frame finishes and when the next frame will be submitted (according to the FPS requirement).

Within the idle period of a game, *PilotFish schedules the computation kernels safely without interfering with the games.* The computation kernels for DL training should only be executed between the end of the previous frame and the start of the next frame. This relies on the kernel duration predictor to provide the execution time of the computation kernels, by leveraging their predictability as we discussed in Figure 6. A computation kernel can be submitted only when it can finish

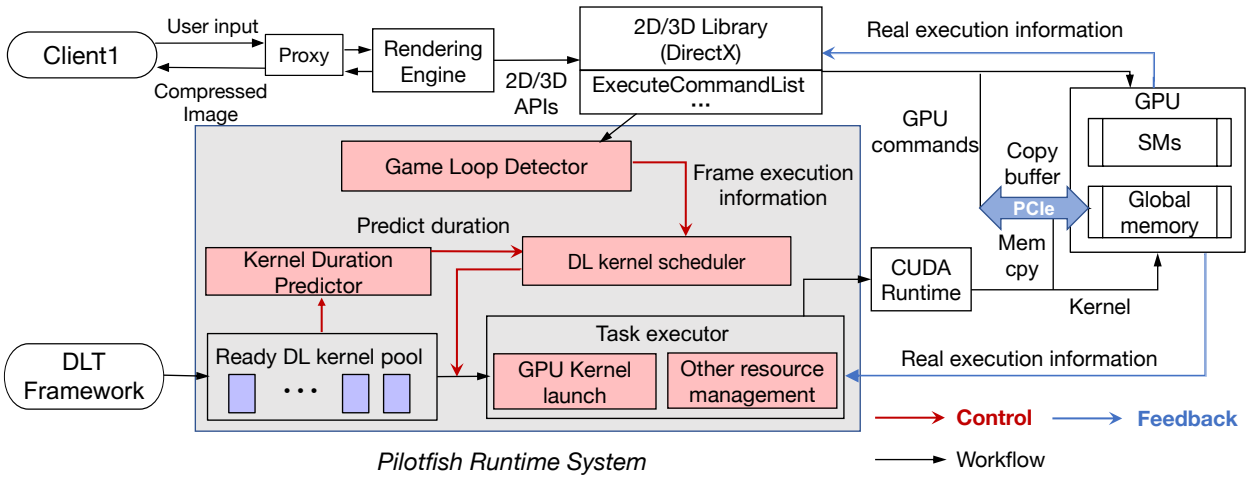


Figure 8: The Overall design of PilotFish. The Game Loop Detector quickly obtains the idle GPU periods via instrumenting the graphic libraries. The DL kernel scheduler dynamically and safely schedules the kernels with the predicted kernel duration. The task executor guarantees the DL kernel execution will not interfere with cloud gaming on GPU and other types of resources.

before the rendering of the next frame starts so that it will not contend with the game rendering on GPU. Since PilotFish only schedules DL kernels without changing its computation, it has no impact on the computation result of DL training.

During the execution of DL training's computation kernels, PilotFish keeps monitoring their progress in its task executor. Once potential interference could appear due to straggler kernels, *PilotFish should immediately preempt the job to guarantee cloud gaming is not affected.* To minimize the loss of training progress due to preemption, we introduce a low-overhead checkpointing mechanism to only kill the computation kernels without losing the trained weight in memory.

We explain in § 4 how PilotFish instruments the graphic libraries to obtain the idle GPU periods. In § 5, we elaborate on how the computation kernels of DL training are scheduled. Then, we demonstrate the task executor in § 6 that manages the task execution on GPUs and other types of resources to provide the strict guarantee of no interference to games.

4 Game Loop Instrumentation

To capture the random idle GPU cycles from games, we need to monitor the frame execution information, i.e., the start and end rendering time of each frame, in real time. Nowadays, there are many popular frame monitoring software for rendering workloads including PresentMon [17], IntelGPA [9], GpuView [8], and FrameView [5]. They all use event-tracing technology [4], which records events with high latency (usually >1 second). However, cloud gaming usually requires 60 frames per second, i.e., 16.67 ms per frame, which cannot accept such a large tracing latency.

In PilotFish, we exploit the fact that most games are developed on common graphic libraries (e.g., DirectX [3], Vulkan [20]), which translate the graphical operations into GPU commands. When the game finishes generating the draw calls, the GPU commands will be submitted to the GPU via a specific API (e.g., ExecuteCommandList in DirectX). PilotFish instruments the command submission API of graphic libraries to detect the start time of frame rendering. The instrumentation latency is very low, usually within 1 microsecond per frame. Moreover, to obtain when the rendering completion time of the submitted frame, PilotFish inserts an additional GPU command for notification of rendering completion at the end of the submission queue. Since the QoS of cloud gaming determines the maximum frame rate, e.g., an FPS of 60 means there is at most one frame per 16.67ms. When PilotFish is notified with the rendering completion, we can calculate when the next frame would appear, thus the time period before the next frame is guaranteed to be idle. Table 2 shows the average overhead and error for FPS perception through the game loop detector. The overhead is negligible where the average overhead per 60 frames is around 0.1ms. We also validate FPS measured from PilotFish by comparing with PresentMon [17] as the ground-truth. The average measurement error of FPS is 0.526%. Instrumenting the graphic libraries that most games built on allows PilotFish to generally support a wide range of existing games and future games without specific modification for every game.

Table 2: The Game Loop Detector Performance.

	Avg. Overhead / 60 frame	Avg. Err.
ACOdyssey	0.1058 ms	0.363%
Genshin Impact	0.070668 ms	0.526%

Algorithm 1: DL training scheduler

```
1  while true do
2      if isFrameRendering() then
3          WaitForFrameComplete();
4          freeTimeslice = FrameTimeQoS -
             LastFrameRenderingTime;
5      else
6          kernel = GetKernelFramePool();
7          kernelTime = PredictDuration(kernel);
8          if freeTimeslice > kernelTime then
9              LaunchKernel(kernel);
10             freeTimeslice =
                 freeTimeslice − kernelTime;
```

5 DL Training Scheduler

With the captured GPU idle periods, PilotFish will schedule the computation kernels from DL training to harvest the free GPU cycles. As shown in Figure 9, PilotFish only allows the DL kernels to execute within the idle GPU periods to avoid GPU contention. Algorithm 1 describes the scheduling strategy of PilotFish: (1) when the game is using GPU to perform rendering, it will wait for the notification of the rendering completion; (2) when the game finishes rendering a frame, the scheduler sends the DL kernels that can finish before the deadline when FPS QoS is affected (e.g., when the QoS is 60 frames per second, the start time between two frames should be no more than 16.67 ms).

PilotFish's DL training scheduler relies on the prediction of computation kernels to decide whether the submitted kernel can finish before the next frame starts (Line 7 in Algorithm 1). PilotFish leverages the predictability and iterative pattern of DL training. The kernels for the same model will be repeatedly submitted in every iteration with different input data. As we have shown in Figure 6, the kernel duration has a very low variance, which can be easily obtained via offline profiling. In PilotFish, the DL training jobs to co-locate with games will be profiled on idle GPUs for tens of iterations (usually a few minutes), and record their kernel execution time.

Note that, the GPU context for DL training is first created in the job initialization, thus its overhead does not affect the scheduling of DL kernels. Also, the launching of computation kernels has an overhead of 10 us, which is usually less or equal to a kernel's execution time. To hide the kernel launching overhead, like most training frameworks, PilotFish submits the computation kernels asynchronously (as shown in Figure 9). Therefore, PilotFish only suffers from at most one kernel launching overhead at the first DL kernel in each frame, which is negligible.

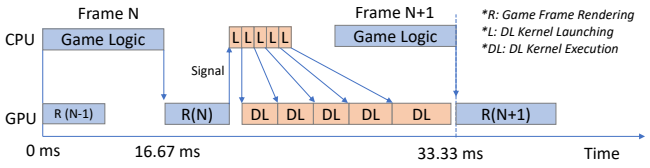


Figure 9: Fine-grained scheduling of DL kernel.

6 Task Executor

After the DL kernel is scheduled, the task executor monitors the kernel execution to avoid straggler kernels that run longer than expected and do not finish before the next rendering frame. In case the potential interference could appear, the task executor will terminate the process quickly to reclaim the GPU for game rendering while minimizing the loss of training progress. In addition to GPUs, the task executor also manages the other resource types including CPUs, PCIe bus and Disk I/O to avoid non-GPU interference.

6.1 GPU Kernel Execution

During the execution of computation kernels, PilotFish's task executor keeps monitoring the running kernels on their execution time. Although, not often, some straggler kernels may run longer than the predicted time, which may postpone the rendering if they do not finish before the next frame appears. Note that the straggler kernels will not lead to QoS violation if the rendering time of the next frame is short and can still be finished within the deadline. Also, a slight drop in FPS ($1 \sim 2$ FPS) may not affect the gaming experience for non-sensitive players. Therefore, PilotFish provides two types of guarantees:

1) Hard guarantee: once a straggler kernel appears that it can not finish before the next frame rendering begins, the task executor suspends the running DL kernel on the GPU immediately.

2) Soft guarantee: PilotFish does not terminate the straggler kernels unless FPS drop exceeds a certain threshold.

Using soft guarantee is more friendly to DL training models that contain kernels of long execution time, e.g., the longest kernel of LSTM runs for 2.4 ms. Our evaluation in § 7.4 shows using the soft guarantee can harvest over 30% more GPU cycles than the hard guarantee when we co-locate LSTM with RDR2.

6.2 Low-overhead Pause and Resume

Figure 10 shows the design of PilotFish's DL training pause and resume. In order to terminate the straggler kernel quickly, PilotFish leverages the multi-priority streams of modern GPUs to send asserting signal to DL training kernels at the highest priority. The preemption can be done very fast within 0.7 ms. However, asserting the kernel would wipe out all the

memory state that results in loss of the training progress. Although DL training may periodically save checkpoints, it is done in a less frequent manner (usually every a few epochs that takes hours).

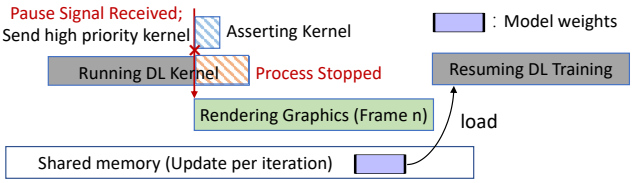


Figure 10: PilotFish's low-overhead pause and resume.

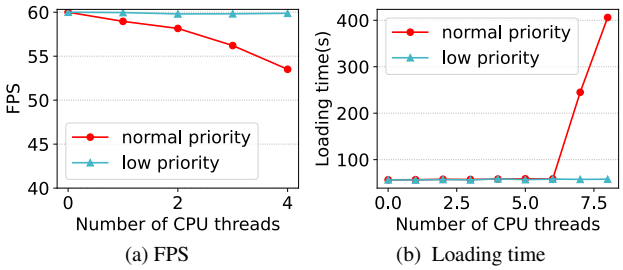
Note that, we only want to terminate the computation to avoid the interference to games thus it is not necessary to clear the memory. To maintain the model weight while DL training job suspension, PilotFish builds a shared memory pool in an isolated process, that stores a backup version of the model weight. When resuming a DL training job from suspension, the pointer of shared memory is directly shipped to the memory manager of the training frame. If the GPU supports inter-process communication (IPC), the shared memory pool is placed on GPU thus no memory copy is needed. Otherwise, The shared memory pool is placed on the host memory thus requires resume the model weights by copying them from host to GPU. Our evaluation shows resuming the model from the host memory for ResNet-34, VGG-16, MobileNet and LSTM takes 64, 69, 63 and 30 ms respectively. But PyTorch's requires over 7 seconds via its default checkpointing mechanism.

6.3 Mitigating Other Resource Contention

In addition to contention on GPU, both cloud gaming and the DL training involve other resource types thus also need careful management for interference avoidance.

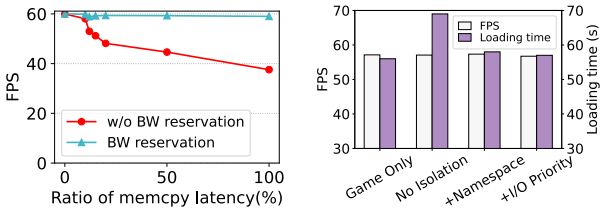
CPU contention. For the DL training tasks, the CPU is used for data pre-processing, e.g., image decoding, re-shaping, data augmentation. Games use CPU for processing game logic and simulate physical effects. CPU contention may appear when the CPU-heavy DL training and games are co-located, resulting in a decrease in FPS and an increase of game loading time. PilotFish solves the resource contention on CPU by setting the priority of threads: game threads use a high priority and DL training threads use a low priority. Figure 11 shows the FPS of RDR2 to be co-located with a job that only pre-processes the data of DL training. By increasing the stress of the co-located job, the FPS and loading time of the game is affected severely if they have the same CPU thread priority. The Windows OS's scheduler can fully mitigate the interference on CPU after we set the thread priority of the co-located job to low.

PCIe contention. Using PCIe bus, games transfer vertex data and primitive data from pageable memory to GPU during execution, and the rendered frames are passed back from GPU [21]. The DL training uses PCIe to transfer data and model parameters. We have tested two popular games' performance benchmarks (Shadow of the Tomb Raider and The Division 2). The average memcpy time per frame is 0.1748ms and the frames with copy time greater than 0.5ms account for 3.9% of all frames. When games use pageable memory and transfer data through PCIe bus alone, the achieved data transfer rate is 11,045MB/s. Because the theoretical peak bandwidth of 16x PCIe 3.0 bus used in our platform is 15,800MB/s and the effective bandwidth is 12,160MB/s, the bus can only support at most $\lfloor \frac{12160}{11045} \rfloor = 1$ memcpy task to transfer data in their full speeds in the same direction. Therefore, it is necessary to guarantee no interference on PCIe to avoid the game's data transfer. In PilotFish, we rely on the bandwidth reservation technique proposed in Baymax [25] to reserve the enough PCIe bandwidth for cloud gaming. The DL training can only transfer data when the game is not using PCIe.



(a) FPS (b) Loading time

Figure 11: The FPS and loading time of RDR2 when co-located with CPU threads for DL training.



(a) PCIe bus (b) Disk I/O

Figure 12: The inference to RDR2 on PCIe bus and Disk I/O.

Figure 12a shows the FPS of game RDR2 when co-located with a stress test progress of memory copy. This stress test copies data from the host memory to the global memory of GPU, and then back to the host memory every 60 ms. We control the proportion of the memcpy time to the total time by controlling the size of the copied data. With increased memory copy stress, the FPS drops greatly without reserving the PCIe bandwidth for the game. The reservation guarantees the game is not affected by PCIe contention.

Disk I/O contention. From disk, games loads rendering

resources (e.g., texture) and DL training loads training data. Contention on disk I/O may lead to longer loading time for games. Figure 12b illustrates the FPS and loading time of a game co-located with a disk stress benchmark performing sequential read/write and 4K read/write [2]. Without any isolation, the FPS does not change but the loading time is increased by 21%. Moreover, we observe some objects are not rendered in the displayed frame, which is unacceptable to players. We apply the widely used I/O isolation techniques, including namespace [12] and I/O priority [10]. We find both techniques can guarantee the game performance by isolating the I/O operations.

GPU memory and caches. To avoid swapping data among GPU memory and host memory, PilotFish only co-locates a game and a DL training job when the sum of their peak GPU memory demand can fit into the GPU memory. Since DL kernels are only executed in the idle GPU cycles, their data movement between GPU memory and GPU caches has no overlap with gaming. GPU commands from game and DL training are serialized without preemption thus there is no context switching overhead. DL training may flush the GPU cache of rendering data of the previous frame. But we do not observe impact on rendering time to the next frame.

Network and video stream encoding. In PilotFish, we assume the distributed training uses a separate network from the cloud gaming service due to security and performance concern, thus there is no interference in network. Also, as we have explained in Figure 2, video stream encoding is done in a separate hardware encoder, thus is not interfered by DL training.

7 Evaluation of PilotFish

We have implemented a prototype of PilotFish on DirectX 12, CUDA 11.1, Windows 10, and PyTorch 1.8 with 2400 lines of code. As far as we know, PilotFish is the first system that co-locate cloud gaming with DL training. Therefore, we compare PilotFish with several straw-man solutions to evaluate its effectiveness. Overall, PilotFish can harvest up to 85% of idle GPU cycles from cloud gaming without generating interference.

7.1 Experimental Setup

We evaluate PilotFish with Steam Remote Play & Steam Link (cloud gaming platform) using the Nvidia RTX 2060 GPU. Table 3 summarizes the software and hardware experimental configurations. Note that PilotFish does not rely on any special hardware features of RTX 2060, and is easy to be set up on other GPUs. As listed in Table 4, we use five popular DirectX 12 games and four DL training applications to perform the experiment.

Throughout our experiments, the FPS target of games is 60 FPS (16.67 ms/frame). The QoS of the game is defined as the 99%-ile latency normalized to 60 FPS. We calculate the GPU utilization as the portion of time when the GPU is busy, which is the same as the definition of nvidia-smi [16]. We define the metric, harvest ratio, as the portion of GPU idle time that is harvested for DL training, which is calculated as

$$\text{Harvest Ratio} = \frac{\text{GPUUtil}_{\text{co}} - \text{GPUUtil}_{\text{Game}}}{100\% - \text{GPUUtil}_{\text{Game}}}, \qquad (1)$$

where $\text{GPUUtil}_{\text{Game}}$ is the GPU utilization of running game independently, and $\text{GPUUtil}_{\text{co}}$ is the GPU utilization when game and DL training are co-located. For PilotFish, the time of model checkpointing is not considered as harvested.

Table 3: Hardware and software specifications.

	Specification
Hardware	Intel(R) i7-7700 @ 3.60GHz Nvidia GeForce RTX 2060
Software	Windows10 19043.1110 CUDA Driver 11.1.96 CUDA SDK 11.1 DirectX 12.1 PyTorch 1.8.1

Table 4: Benchmarks used in the experiment.

Benchmarks	Workloads
Ashes of the Singularity (AOS)	Crazy quality on2560*1440; FPS: 60 GPU focused benchmark
Red Dead Redemption 2 (RDR2)	Favor performance quality on 2560*1440; FPS: 60
Shadow of the Tomb Raider (SOTTR)	High quality on 2560*1440; FPS: 60
F1 2021 (F1)	Medium quality on 1920*1080; FPS: 60
HITMAN3 (HIT3)	Ultra quality on 2560*1440; FPS: 60
DL Training	ResNet-34 (RS) [30]; VGG-16 [44] ; MobileNet (MN) [31]; LSTM [45]; Dataset: ImageNet-1k, Wikitext-2

Comparison Baselines. To compare the performance of PilotFish, we propose three straw-man solutions:

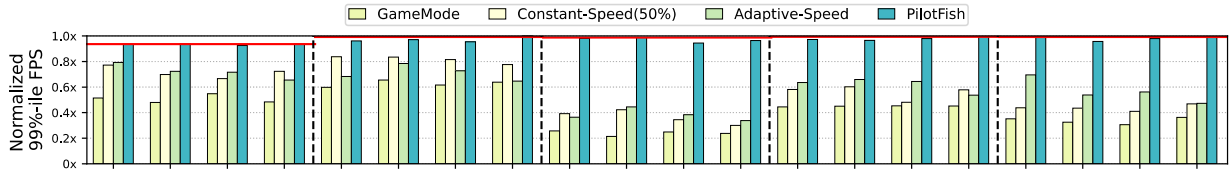
1. **GameMode** [6] is a feature introduced by Windows to prioritize CPU threads of games. It does not control GPU execution.

2. **Constant-Speed** controls the DL kernel submission speed with a constant rate.

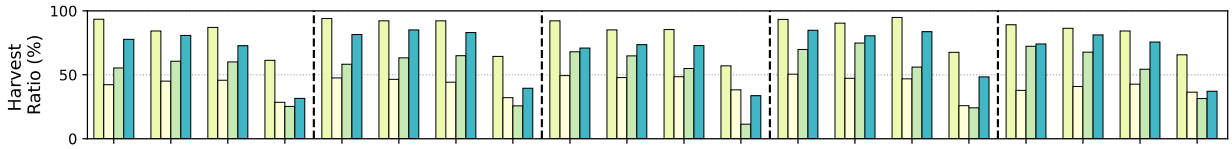
3. **Adaptive-Speed** controls the DL kernel submission speed dynamically according to the FPS profiled from the event-tracing tool PresentMon [17]. If FPS < 60, the DL kernel submission speed is halved, otherwise, it is multiplied by 1.2.

7.2 GPU Utilization Improvement and FPS Guarantee

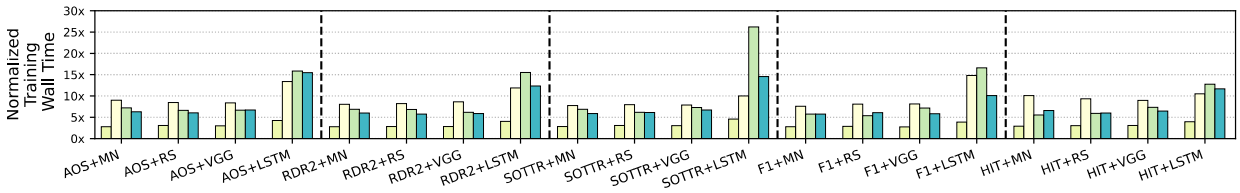
We first demonstrate the effectiveness of PilotFish by comparing PilotFish with the three baselines on all combinations of cloud games and DL models listed in Table 4. By default, the Constant-Speed baseline is set to 50% of the ideal speed (i.e., training the model on the same GPU without co-location).



(a) The 99%-ile FPS normalized to the FPS target (60 FPS). The red line shows the 99-tile FPS of running each game without co-location.



(b) The harvest ratio of idle GPU time of cloud games.



(c) The training wall time of co-located job normalized to training on dedicated GPU system

Figure 13: The 99%-ile FPS, harvest ratio and training wall time of different co-location combinations of games and DL models.

Figure 13 presents the 99%-ile FPS of the cloud games normalized to the FPS target (60 FPS), and the harvest ratio of idle GPU time. Note that, due to bursty complex frames, the cloud game may not always maintain at 60 FPS either even without co-location. Figure 13a shows that PilotFish achieves almost the same 99%-ile FPS compared to that without co-location. The three baselines all experienced severe FPS drops. GameMode drops the most, by up to 78.6% (e.g. SOTTR+RS). Constant-Speed(50%) and FPS-Based drop from 16.3% to 69.2% and from 20.7% to 66.3%, respectively. In the game of SOTTR and HIT, all baselines suffer from severe interference. Since SOTTR switches scenes multiple times during the benchmark, its rendering time of frames fluctuates more severely than the other four games. The three baselines cannot quickly adapt to the fluctuation thus perform poorly.

Figure 13b shows the harvest ratio in the different combinations. As shown in the figure, PilotFish w/ hard guarantee harvests 78.56% of idle times on average in all five games co-located with three DL training tasks (MobileNet, Resnet-34 and VGG-16) without interfering with the cloud games. When cloud games are co-located with LSTM, the harvest ratio drops to 39.03%. Because LSTM contains some large kernels that run for $\sim$ 2.4 ms, they may not be scheduled if the idle GPU time is short with PilotFish's hard guarantee. Since the rendering time of game F1 is lower than other games, its harvest ratio on LSTM is relatively higher than others, which is 48.43%. With the huge penalty of FPS drop, GameMode achieves the highest harvest ratio (83% on average) since it does not control the speed of DL training. The harvest ratios of the Constant-Speed (50%) and Adaptive-Speed range from 26% to 50% and 11% to 74%. These two baselines not only harvest less idle GPU time than PilotFish but also degrades the FPS significantly. They prove the necessity of PilotFish's mechanisms to fast and safely schedule DL kernels.

Figure 13c shows the training wall time of the co-located DL models normalized to training them on dedicated GPU. The training wall time is almost inversely proportional to the harvest ratio. Because GameMode occupies more GPU cycles from games in addition to the idle cycles, it has the least slowdown at the price of severely affected game FPS. Because of higher harvesting efficiency, PilotFish's training wall time is better than Constant-Speed and Adaptive-Speed for most models without affecting the FPS of games.

7.3 Dissecting Execution

To demonstrate how cloud game runs when co-located with DL training, in Figure 14, we show the instantaneous FPS (the inverse of frame time) fluctuation of RDR2 over time when co-located with ResNet-34. We select a game segment (50 seconds) during the stable running of the game. We find PilotFish can always be stable near the original FPS without co-location. The baselines experience serious FPS fluctuations, especially GameMode and Constant-Speed since they are not

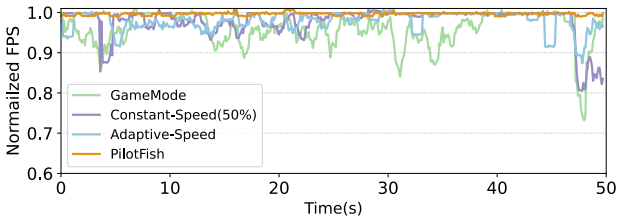


Figure 14: The instantaneous FPS of RDR2 over time when co-located with ResNet-34. The FPS is normalized to the average FPS without co-location.

Figure 15: The rendering quality of PilotFish (left), No co-location (middle), and GameMode (right). The rendering quality in GameMode is much worse than the others due to interference.

adaptive at all. The fluctuated and degraded FPS leads to very poor experience for players. When FPS drops, some games with adaptive rendering mechanism will actively reduce the rendering quality to maintain a smooth playing experience. Figure 15 compares the rendering quality of PilotFish, no co-location, and GameMode. PilotFish has the same rendering quality with running the game without co-location. But the game co-located with DL training in GameMode reduces the rendering quality under the bridge due to interference.

7.4 Sources of Improvement

Dynamic Scheduling. Figure 16 shows normalized 99%-ile FPS and the harvest ratio of the pair (RDR2+RS) under the different kernel submission speed in Constant-Speed policy. The kernel submission speed ranges from 3% to 100% (normalized to the ideal speed without co-location). The rightmost column shows the results of PilotFish for comparison. As expected, we find that the 99%-ile FPS decreases and the harvest ratio increases as the submission speed grows from 3% to 100%. We specifically listed the 99%-ile FPS at the kernel submission speed 3% and 4%. We find the FPS target can be satisfied only when the kernel submission speed is very low. When the submission speed is higher than 4%, 99%-ile FPS begins to drop. Without degrading the 99%-ile FPS, PilotFish can achieve the same harvest ratio of Constant-Speed at 80% submission speed.

Figure 17 shows the harvest ratio of HIT under different rendering qualities when co-located with MobileNet and LSTM.

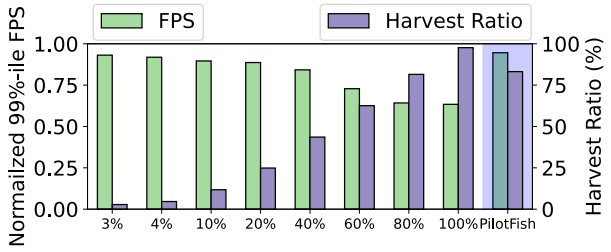


Figure 16: The impact of kernel submission speed in Constant-Speed v.s. PilotFish.

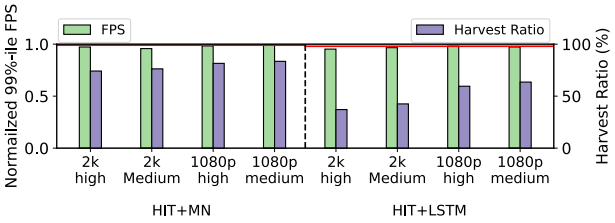


Figure 17: (HIT+MN/LSTM) the 99%-ile FPS and harvest ratio of PilotFish with different graphic quality.

Since MobileNet is mainly comprised of small kernels, it is easier to fit into short idle GPU periods. But the LSTM model contains some long kernels ($\sim$ 2.4 ms), which requires longer idle GPU periods. Therefore, reducing the rendering quality allows LSTM to harvest more GPU cycles from the game than MobileNet.

The two experiments in Figure 16 and Figure 17 imply the dynamic scheduling of DL kernels is necessary to handle the high randomness of game frames and diverse characteristic of different combinations of game and DL model.

Effective training pause and resume. To verify the need for the training pause mechanism introduced in Section 6.1, we disable this mechanism to compare its impact with PilotFish's policies of hard guarantee and soft guarantee. Figure 18 shows the 99%-ile FPS (normalized to 60 FPS) and the harvest ratio with different pause policies. When using the policy of hard guarantee, PilotFish achieves the same FPS of that without co-location. When the pause condition is relaxed by 5% (3 FPS), the FPS using the policy of soft guarantee is degraded within the threshold while the harvest ratio is increased. When disabling the pause mechanism, the FPS further decreases at the cost of no FPS guarantee. The impact of the pause policies is different for models: ResNet-34 is less impacted than LSTM since the computation kernels of ResNet-34 is much shorter than LSTM. In the worst case, if a DL training task submits a long-running kernel, the game rendering could be infinitely postponed. Therefore, we suggest using the policy of soft guarantee when the cluster opera-

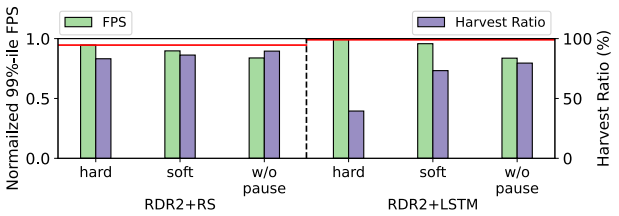


Figure 18: The 99%-ile FPS and harvest ratio of PilotFish with different training pause polices.

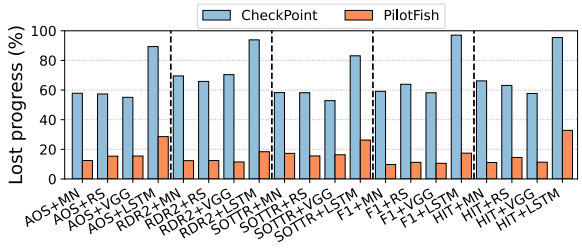


Figure 19: (RDR2+RS) the lost progress of DL training.

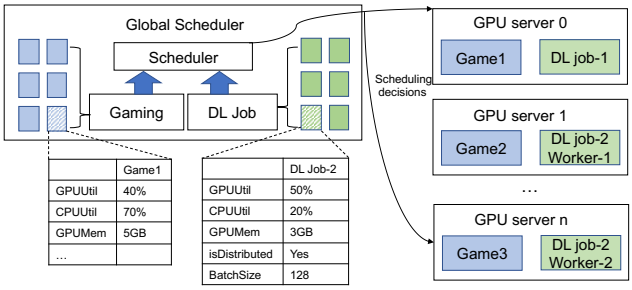


Figure 20: Cluster-level Scheduler.

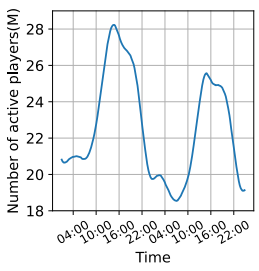


Figure 21: The variation of active players on Steam.

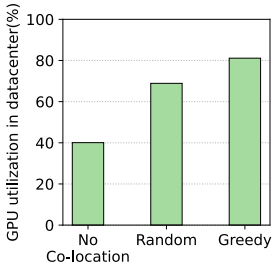


Figure 22: The GPU utilization in the simulated cluster.

tor wants to trade a limited interference with a higher GPU utilization.

Figure 19 shows the lost progress of DL training due to training pause with the hard guarantee. Compared with the epoch-level checkpoint in PyTorch whose lost progress is 69.62% on average, PilotFish reduces the lost progress by 4.6 times to 15.04% with the weight backup in the shared memory pool. LSTM loses more training progress than other models since it triggers more training pause due to its longer computation kernel.

8 Scale to Data Center

To evaluate the potential benefit of PilotFish to cloud gaming service running in a large cluster of GPUs, we use a simple heuristic cluster-level scheduler to decide which games and DL training jobs are suitable for co-location on the same GPU, as shown in Figure 20. The heuristic cluster scheduler collects the average resource usage of DL jobs and games through offline profiling. It greedily matches the DL training job with the game with the DL training job so that the remaining resource is minimized. For the DL model using synchronous data parallel training, the scheduler prefers to deploy each of its workers to the servers with a similar utilization so that each worker can run at a similar speed, which can reduce the synchronization overhead.

We compare this heuristic policy with a random scheduling policy that co-locates games and DL training models randomly. We simulate a cloud gaming cluster composed of one thousands Nvidia RTX2060 GPUs. We select ten popular games as the workload of cloud gaming, including Dota 2, League of Legends, PUBG, CS:GO, Civilization 5, Assassin's Creed Odyssey, The Division 2, Ashes of the Singularity, RDR2, and Genshin Impact. The games are launched with the same probability. The number of running games follows the active player variation reported by Steam (shown in Figure 21), which has a strong diurnal pattern. We regard the peak point in Figure 21 as the situation when all the 1000 GPUs are used by cloud games. For the DL Training workloads, we select 750 instances evenly from the five models: ResNet-34, ResNet-50 [30], VGG-16 [44], MobileNet [31], and DenseNet [32]. Each model has 100 non-distributed training instances and 50 distributed training instances.

Figure 22 shows the GPU utilization of no co-location, random scheduler and the greedy heuristic scheduler. When there is no co-location, the cloud gaming cluster only has a GPU utilization of $\sim$ 40%. The random policy can improve the utilization to 68.89% due to PilotFish's efficient execution. Since the greedy policy is aware of the resource usage pattern of cloud games and DL training, it can further improve the cluster utilization to 81.12%. It implies the games and DL training jobs should be carefully scheduled at the cluster-level to maximize the benefit of co-location, which is an interesting future direction.

9 Related Work

General CPU co-location. There has been a large amount of prior work focusing on improving application QoS and

hardware utilization for CPU co-location. They can be broadly categorized as (1) profiling-based methods [26, 27, 41, 48, 52] and (2) partitioning-based methods [39, 53]. The profiling-based methods, such as Bubble-Up [41], Bubble-Flux [48], SMiTe [52], uses offline profiling of user-facing services and batch applications to predict their performance degradation to avoid contention on shared cache and memory bandwidth. They periodically adjust the allocation of shared resources according to the QoS feedback of user-facing services.

However, these techniques would fail on cloud gaming because they neglect the complex interaction of interference on different shared resources on GPUs.

General GPU co-location. Several techniques were proposed in the prior work to improve the utilization of GPUs with co-location. TimeGraph [34] and GPUSync [28] use priority-based scheduling to guarantee the performance of real-time kernels. High-priority kernels are executed first if multiple kernels are launched to the same GPU. GPU-EvR [35] launches different applications to different streaming multiprocessors (SMs) on one GPU. However, they are not applicable to our problem because they all rely on the simulator to synthesize the execution trace of co-located applications. Laius [50] and Baymax [25] predict the kernel duration and reorder the kernel based on the QoS headroom of user-facing queries. But it is difficult to predict the rendering frame time of game with a low overhead. AntMan [47] only co-locates multiple DL training jobs, which cannot handle the unpredictable game rendering. Nvidia Volta MPS (Multi-Process Service) [42] enables multiple applications to share a GPU concurrently with static partition, however, cannot handle the dynamic load of cloud gaming. Moreover, MPS-based solutions rely on the special hardware feature that only supports CUDA applications but not games, and is not applicable to non-Nvidia GPUs.

Co-location of cloud gaming. Specifically for cloud gaming, several works have been proposed to improve resource utilization by co-locating multiple games [37, 43, 49]. vGASA [49] adaptively schedules rendering tasks from multiple games to meet the SLA in a best-effort manner. However, when a hard SLA guarantee is required, vGASA has to reserve the resource for the worst cases so that all running games can meet the SLA at the most complex scenes. As we have shown in Section 2.2, cloud gaming has a high variance in GPU usage. Conservatively guaranteeing the worst case would waste resources with a significant over-provisioning.

GAugur [36] and dJay [29] dynamically tune the game settings for the co-located games during gameplay to adapt to changes of game scenes for improving performance. However, as we have shown in Figure 4, the frame time and GPU load in the gaming could fluctuate drastically even within a short period of time. Frequently changing the game setting is noticeable to players and could greatly degrade the gaming experience. This is unacceptable to commercial cloud gaming services. Instead, the computation of DL training is highly predictable. PilotFish can accurately predict the execution of DL kernels and schedule them only when it is safe. This is the main reason why we claim DL training is the right workload to be co-located with cloud gaming.

10 Conclusion

Cloud gaming service suffers from low GPU utilization issue due to the limitation of network and edge devices. Since cloud gaming utilizes GPUs in a very random manner, existing co-location solutions for GPU cannot meet the QoS requirement of cloud gaming. PilotFish addresses this issue by co-design cloud gaming service and deep learning training framework. PilotFish can harvest free GPU cycles using DL training with no interference to cloud gaming. PilotFish achieves this hard guarantee by (1) quickly capturing the idle GPU periods from cloud gaming via low-overhead instrumentation to graphic libraries (e.g., DirectX); (2) leveraging the predictability of DL computation to safely schedule DL kernels; and (3) providing a low-overhead mechanism to pause DL computation when they could potentially interfere with games. Our evaluation shows that PilotFish can harvest a significant portion of idle GPU time of cloud gaming up to 85.1% without affecting the gaming experience. PilotFish reveals a principled design to co-locate unpredictable workloads with predictable low-priority workloads. In addition to co-locating cloud gaming with DL training, it is interesting to generalize PilotFish's solution on other predictable workloads, e.g., scientific computing [22, 50].

Acknowledgments

This work is partially sponsored by the National Natural Science Foundation of China (62022057, 61832006, 61872240), and Shanghai international science and technology collaboration project (21510713600). We thank the anonymous reviewers for their constructive feedback and suggestions. Zhenhua Han, Quan Chen and Minyi Guo are the corresponding authors.

References

[1] Amazon appstream. http://aws.amazon.com/appstream.

[2] As ssd benchmark. https://www.alex-is.de/PHP/fusion/news.php.

[3] Directx. https://docs.microsoft.com/en-us/windows/win32/directx.

[4] Event tracing for windows. https://docs.microsoft.com/en-us/windows/win32/etw/about-event-tracing.

[5] Frameview. https://www.nvidia.com/en-us/geforce/technologies/frameview/.

[6] Gamemode. https://support.xbox.com/en-US/help/games-apps/game-setup-and-play/use-game-mode-gaming-on-pc.

[7] Google stadia. https://stadia.google.com.

[8] Gpuview. https://docs.microsoft.com/en-us/windows-hardware/drivers/display/using-gpuview.

[9] Intelgpa. https://software.intel.com/content/www/cn/zh/develop/tools/graphics-performance-analyzers.html.

[10] I/o prioritization in windows os. https://clightning.medium.com/i-o-prioritization-in-windows-os-6a0637874a52.

[11] Microsoft xbox remote play. https://www.xbox.com/en-US/consoles/remote-play.

[12] Namespace. https://docs.microsoft.com/en-us/windows/win32/adsi/namespaces.

[13] Nvenc. https://developer.nvidia.com/nvidia-video-codec-sdk.

[14] Nvida cuda. https://developer.nvidia.com/zh-cn/cuda-toolkit.

[15] Nvidia geforce now. https://www.nvidia.com/en-us/geforce-now/.

[16] Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface.

[17] Presentmon. https://github.com/GameTechDev/PresentMon.

[18] Sony playstation now streaming. http://us.playstation.com/playstationnow.

[19] Steam survey. https://store.steampowered.com/stats/Steam-Game-and-Player-Statistics.

[20] Vulkan. https://www.vulkan.org/.

[21] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. A survey on cloud gaming: Future of computer games. *IEEE Access*, 4:7605–7620, 2016.

[22] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee, 2009.

[23] Quan Chen, Zhenning Wang, Jingwen Leng, Chao Li, Wenli Zheng, and Minyi Guo. Avalon: towards qos awareness and improved utilization through multi-resource management in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*, pages 272–283, 2019.

[24] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 17–32, 2017.

[25] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. *ACM SIGPLAN Notices*, 51(4):681–696, 2016.

[26] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, volume 48, pages 77–88. ACM, 2013.

[27] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[28] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time gpu management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44. IEEE, 2013.

[29] Sergey Grizan, David Chu, Alec Wolman, and Roger Wattenhofer. djay: Enabling high-density multi-tenancy for cloud gaming servers with dynamic cost-benefit gpu load balancing. In *Proceedings of the sixth ACM symposium on cloud computing*, pages 58–70, 2015.

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[31] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[32] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869*, 2014.

[33] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. Analysis of large-scale multi-tenant {GPU} clusters for {DNN} training workloads. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 947–960, 2019.

[34] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[35] Haeseung Lee, Al Faruque, and Mohammad Abdullah. Gpu-evr: Run-time event based real-time scheduling framework on gpgpu platform. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 220. European Design and Automation Association, 2014.

[36] Yusen Li, Chuxu Shan, Ruobing Chen, Xueyan Tang, Wentong Cai, Shanjiang Tang, Xiaoguang Liu, Gang Wang, Xiaoli Gong, and Ying Zhang. 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*, pages 231–242, 2019.

[37] Yusen Li, Changjian Zhao, Xueyan Tang, Wentong Cai, Xiaoguang Liu, Gang Wang, and Xiaoli Gong. Towards minimizing resource usage with qos guarantee in cloud gaming. *IEEE Transactions on Parallel and Distributed Systems*, 32(2):426–440, 2020.

[38] Tianyi Liu, Sen He, Sunzhou Huang, Danny Tsang, Lingjia Tang, Jason Mars, and Wei Wang. A benchmarking framework for interactive 3d applications in the cloud. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 881–894. IEEE, 2020.

[39] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.

[40] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 881–897, 2020.

[41] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259. ACM, 2011.

[42] NVIDIA. Sharing a gpu between mpi processes: multi-process service(mps). Oct. 2012.

[43] Zhengwei Qi, Jianguo Yao, Chao Zhang, Miao Yu, Zhizhou Yang, and Haibing Guan. Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(2):1–25, 2014.

[44] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[45] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

[46] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 595–610, 2018.

[47] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. {AntMan}: Dynamic scaling on {GPU} clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 533–548, 2020.

[48] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 607–618. ACM, 2013.

[49] Chao Zhang, Jianguo Yao, Zhengwei Qi, Miao Yu, and Haibing Guan. vgasa: Adaptive scheduling algorithm of virtualized gpu resource in cloud gaming. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):3036–3045, 2013.

[50] Wei Zhang, Weihao Cui, Kaihua Fu, Quan Chen, Daniel Edward Mawhirter, Bo Wu, Chao Li, and Minyi Guo. Laius: Towards latency awareness and improved utilization of spatial multitasking accelerators in datacenters. In *Proceedings of the ACM International Conference on Supercomputing*, pages 58–68, 2019.

[51] Wei Zhang, Kaihua Fu, Ningxin Zheng, Quan Chen, Chao Li, Wenli Zheng, and Minyi Guo. Charm: Collaborative host and accelerator resource management for gpu datacenters. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 307–315. IEEE, 2021.

[52] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 406–418. IEEE, 2014.

[53] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 33–47, 2016.