Maximizing the Utilization of GPUs Used by Cloud Gaming through Adaptive Co-location with Combo

Binghao Chen, Han Zhao, Weihao Cui, Yifu He, Shulai Zhang,
Quan Chen, Zijun Li, Minyi Guo
chenbinghao@sjtu.edu.cn,zhao-han@cs.sjtu.edu.cn,{weihao,yifu.ho,zslzsl1998}@sjtu.edu.cn
chen-quan@cs.sjtu.edu.cn,lzjzx1122@sjtu.edu.cn,guo-my@cs.sjtu.edu.cn
Shanghai Jiao Tong University
China

ABSTRACT

Cloud vendors are now providing cloud gaming services with GPUs. GPUs in cloud gaming experience periods of idle because not every frame in a game always keeps the GPU busy for rendering. Previous works temporally co-locate games with best-effort applications to harvest these idle cycles. However, these works ignore the spatial sharing of GPUs, leading to not maximized throughput improvement. The newly introduced RT (ray tracing) Cores inside GPU SMs for ray tracing exacerbate the situation.

This paper presents Combo, which efficiently leverages two-level spatial sharing: intra-SM and inter-SM sharing, for throughput improvement while guaranteeing the QoS of rendering games' frames. Combo is novel in two ways. First, based on the investigation of programming models for RT Cores, Combo devises a neat compilation method to convert the kernels that use RT Cores for fine-grained resource management. We utilize the fine-grained kernel management to construct spatial sharing schemes. Second, since the performance of spatial sharing varies with the actual co-located kernels, two efficient spatial sharing schemes are proposed: exact integrated SM sharing and relaxed intra-SM sharing. In order to maximize the throughput of BE applications, Combo identifies the best-fit scenarios for these two schemes by considering runtime rendering load. Our evaluation shows Combo can achieve up to 38.2% (14.0% on average) throughput improvement compared with the state-of-the-art temporal-only solution.

Han Zhao, Quan Chen and Minyi Guo are the corresponding authors.

CCS CONCEPTS

• **Computer systems organization** → *Real-time system architecture.*

KEYWORDS

Cloud gaming, GPU utilization, Adaptive co-location

1 INTRODUCTION

The increasing prevalence of cloud gaming has attracted the attention of researchers. Cloud gaming players only need a client to receive rendered frames from the cloud server, enabling more players to enjoy excellent gaming effects brought by the powerful GPUs. Therefore, cloud games have become an important workload for cloud providers, such as Microsoft's Xbox Remote Play [6], Amazon's AppStream [1] and Nvidia's Geforce Now [9]. Since mainstream cloud games have a low utilization on game-oriented GPUs (e.g., Nvidia RTX 3090 [5]), previous works try to improve the GPU utilization by co-locating cloud gaming applications and best-effort applications (BE applications). Pilotfish [37], for instance, leverages deep learning (DL) training jobs to harvest the free GPU cycles.

Figure 1 shows the workflow of Pilotfish. In the figure, game frames are rendered at a deterministic frequency (`fps`), thus setting a QoS target of `1/fps` for rendering a single frame. The GPU for gaming (e.g., RTX3090) comprises multiple isomorphic streaming multiprocessors (SMs), and each SM contains multiple computing units: RT Cores (only for ray tracing), CUDA Cores (for general purpose) and Tensor Cores (dedicated for matrix-multiplication). Ray tracing is an important visual effect task in gaming. Games commonly first uses CUDA Cores for traditional rendering, and then perform ray tracing through RT Cores. The GPU kernels for
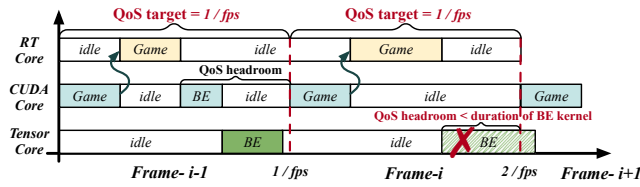
Figure 1: Active timeline of RT/CUDA/Tensor Cores when PilotFish co-locates a game and BE task.

rendering can finish ahead of the QoS target, leaving idle intervals between adjacent frames (QoS headroom). Pilotfish adopts temporal sharing to schedule BE kernels, running either on CUDA Cores or Tensor Cores, to occupy these free GPU cycles. However, such a co-location with only temporal sharing suffers from two problems.

Unexploited intra-SM parallelism. In Figure 1, we observe that only one of the three cores is activated at the same time in temporal sharing. Although, these three computing units reside inside the same SM, our investigations in §2.3 show that without resource contention (thread slots, shared memory), they can be active simultaneously for different computations. During ray tracing, temporal sharing only uses RT Cores, leaving both CUDA Cores and Tensor Cores idle. Since ray tracing accounts for 36.5% of the total rendering time, the long duration of ray tracing makes things worse. The temporal sharing scheme is unable to utilize the intra-SM parallelism for higher throughput.

Under-utilized free GPU cycles. Another observation from Figure 1 is that many free GPU cycles still cannot be utilized in temporal sharing. Both the rendering kernels and the BE kernels such as DL training operators can be compute-intensive. In this case, some BE kernels' duration exceeds the QoS headroom, thus leaving the free GPU cycles un-utilized. However, we can limit the number of SMs allocated for the rendering kernels while ensuring the QoS of rendering frames in a just-right way. The saved SMs then can be utilized by BE kernels for throughput improvement. There is no need to find a BE kernel to fit in the QoS headroom. This is referred to as inter-SM parallelism.

In this paper, we propose COMBO, a high-performance scheduling system that adaptively exploits the two-level sharing scheme: intra-SM and inter-SM sharing, to improve system throughput while ensuring the QoS of cloud games. Specifically, while ensuring QoS of frames in cloud gaming, COMBO switches the GPU sharing schemes according to the cloud game's load mode to maximize the system throughput.

To achieve the above goal, the primary objective of COMBO is to enable spatial sharing of kernels at the level of both intra-SM and inter-SM. Our key insight is that this can be achieved by fine-grained resource management of kernels (especially SM allocations). With SM isolation between different kernels,

games and BE applications use different SMs concurrently (inter-SM). With scheduling blocks[1] from different kernels to the same SM, different computing cores are activated simultaneously (intra-SM). Previous works resolved similar fine-grained management of CD kernels (running on CUDA Cores) and TC kernels (running on Tensor Cores) through Persistent-Thread-Block (PTB) method [23, 32, 39, 40]. However, it cannot be directly applied to the RT kernels (running on RT Cores), which use a different programming language (Optix) [10]. Under these circumstances, COMBO identifies a neat mapping between RT kernels and common GPU kernels. Through the mapping, the RT kernel is then adapted to the PTB version for fine-grained management.

With the converted kernels, COMBO constructs spatial sharing schemes with QoS guarantee by analyzing kernel sharing performance. The experimental results in §4.2 show that the co-location of CD kernels with either TC kernels or RT kernels brings an average 18.1% throughput improvement, but only marginal gain for the co-location between TC and RT kernels. Based on two kernel co-locations with performance gains, two schemes are drawn up.

The first scheme is exact integrated SM sharing, which mainly utilizes the parallelism of TC kernels and CD kernels. Exact integrated SM sharing allocates the best-fit SM number for rendering kernels to exactly satisfy the QoS requirement. BE kernels could utilize the remaining SMs for intra-SM parallelism. The second one is relaxed intra-SM sharing, which mainly utilizes the parallelism of RT kernels and CD kernels. In relaxed intra-SM sharing scheme, only intra-SM sharing is enabled between RT kernels and CD kernels when COMBO can ensure QoS. Otherwise, the relaxed intra-SM sharing scheme falls back to the temporal sharing scheme.

COMBO employs both these two sharing schemes to achieve optimal throughput improvement. Another key insight is that neither of the two schemes could efficiently handle all situations, although they all could satisfy the frame's QoS requirement. Analysis in §5.3 reveals that while exact integrated SM sharing maximizes the performance of BE applications under a stable rendering workload, relaxed intra-SM sharing is able to provide better performance when facing rendering workload fluctuation. Therefore, COMBO predicts the duration of kernels and selects the best-fit sharing scheme at runtime to co-locate games with BE applications.

COMBO further proposes a headroom merging mechanism based on the observation that each frame does not need to be rendered immediately. COMBO just needs to ensure that the frame is rendered before the QoS target. This implies that the free GPU cycles of the adjacent two frames can be merged by delaying the rendering of the latter frame. COMBO enlarges

---

[1]GPU kernel is executed as a grid of blocks, with each block being scheduled onto an SM for computation.

the headroom for scheduling BE kernels with appropriate headroom merging. It should be noted that headroom merging only works for relaxed intra-SM sharing scheme, while inter-SM sharing scheme has almost no headroom.

We have implemented a prototype of COMBO to support the co-location of games with BE applications such as scientific computing and DL training. We evaluate COMBO using popular games for cloud gaming and widely-used best-effort applications. Evaluation results proves that COMBO can strictly guarantee the QoS of cloud gaming when co-located with all BE applications. COMBO could improve the throughput of the co-located BE applications by 14.0% compared with Pilotfish on average (up to 38.2%).

The key contributions of the paper are as follows:

- **We identify the low GPU utilization of the temporal sharing scheme in the cloud gaming co-location scenario.** The root cause is its unawareness of spatial sharing opportunities: intra-SM and inter-SM parallelism.
- **We propose the source-to-source compilation method of the RT kernels, which supports resource management using the PTB method.** Based on this, the RT kernels could enjoy intra-SM or inter-SM parallelism with precise scheduling schemes.
- **We clarify the best-fit scenarios for the proposed inter-SM and intra-SM sharing schemes.** We then design a scheduling mechanism that selects the appropriate mechanism to maximize system throughput based on the cloud gaming load.

2 BACKGROUND AND MOTIVATION

2.1 Cloud gaming and ray tracing

While traditional games require the installation of rendering engines locally, cloud games acquire rendered frames from cloud servers by streaming services. Cloud gaming is advantageous because it eliminates the need to purchase expensive equipment (GPUs). Consequently, cloud gaming is available on a broad spectrum of computing devices, including home computers, tablets, smartphones, and more.

Ray tracing is an important task in rendering scenes. It enables global illumination, ambient light occlusion, motion blur, and other effects that bring the rendered scene closer to reality for cloud gaming users. Achieving such visual effects leads to a significant drop in the frame rate on traditional GPUs. To support real-time ray tracing, Nvidia introduced RT cores in 2018 [13]. RT Cores accelerate Bounding Volume Hierarchy (BVH) traversal and ray casting functions, which are the core computation in the ray tracing process. This results in a 10× faster ray tracing than CUDA Cores. With the Tensor Cores introduced for DL jobs, the newest GPUs for cloud gaming are now equipped with three types of cores.

Table 1: Specifications of an Nvidia RTX 3090 GPU.

Resource	Value	Resource	Value
Number of SMs	82	Max Threads per SM	1024
RT Cores per SM	1	Tensor Cores per SM	4
CUDA Cores per SM	128	Shared Memory per SM	64 KB

Table 2: Configurations of Cloud games.

Cloud games	Configurations
Served Steel (STEEL)	High quality: 2560*1440; FPS: 120
Deliver Us The Moon (DUTM)	High quality: 2560*1440; FPS: 120
Quake2(QUAKE)	High quality: 2560*1440; FPS: 120
The Ascent(ASCENT)	High quality: 2560*1440; FPS: 120

In this paper, we explore the capability of utilizing all computing units inside the SM for throughput improvement. Throughout the paper, we use an Nvidia RTX 3090 GPU as the experimental platform. Table 1 lists the detailed hardware specification of the experimental platform. As indicated in the table, the RTX 3090 comprises 82 RT Cores distributed across 82 SMs. Additionally, each SM has 4 Tensor Cores and 128 CUDA Cores. It is not efficient to disregard the computational ability of Tensor Cores and CUDA Cores entirely, even if a kernel uses the RT Cores efficiently. Temporal sharing employed by previous works [37] treats the GPU as a whole without considering the utilization of SMs and the cores within each SM.

2.2 Problems of Temporal Sharing

To elaborate on the problems of temporal sharing schemes, we conduct an experiment to study the GPU utilization when co-locating the kernels of a cloud game and a DNN training task onto a modern GPU that integrates RT Cores, CUDA Cores, and Tensor Cores.

We choose Pilotfish [37] to exploit the free GPU cycles by co-locating cloud game with BE application while ensuring the QoS of the cloud game. We use four mainstream cloud games(STEEL [14], DUTM [3], Quake2 [12], Ascent [2]) and four mainstream DNN training jobs (Resnet50, VGG16, Inception, Densenet) in this Experiment. Table 2 lists the detailed configurations of the cloud games. Each kernel's duration is collected to compute the duration of all the RT Core kernels (RT kernel), all the CUDA Core kernels (CD kernel), and all the Tensor Core kernels (TC kernel). These three duration values are normalized to the QoS target.

Computing units are activated separately. Figure 2 shows the duration results of different co-located application pairs. While the black portion indicates the duration of all the RT kernels, the grey portion indicates the duration of all the CD kernels, and the red portion indicates the duration of all the TC kernels. We stack the results to show the overall active time of three computing cores. From the figure, we
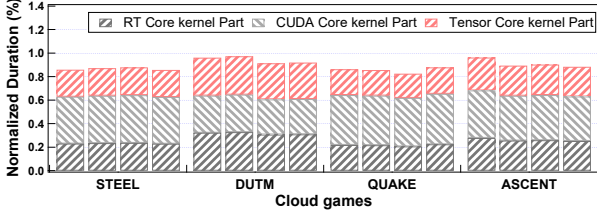
Figure 2: The active time of the kernels with Pilotfish.

observe that the computing units' overall active time does not exceeds the QoS target. This implies that none of these cores are active simultaneously.

Some headroom is not harvested by BE jobs. Another observation from Figure 2 is that there are an obvious gap between the overall active time and the QoS target for all co-located application pairs. We further count the ratio of un-utilized GPU cycles. Experimental results show that the unused cycles account for an average of 10.6% of the QoS target and a maximum of 17.4%. There are many under-utilized GPU cycles under the temporal sharing scheme.

From this experiment, we can conclude that the current temporal sharing scheme supports only the application that is occupying the GPU exclusively. This always leaves two types of the computing resources in an idle state. Meanwhile, the current temporal sharing scheme has many under-utilized free cycles, which can be attributed to the inability to find an appropriate kernel to fit in the QoS headroom.

2.3 Opportunities

A GPU comprises multiple isomorphic SMs, and each SM contains RT Cores, CUDA Cores and Tensor Cores. Apart from the temporal sharing scheme, the spatio sharing schemes also have the potential to improve the overall throughput.

2.3.1 Intra-SM sharing. We first study the potential of utilizing different computing cores on SM in parallel. We implement $Kernel_{RT}$ to be the RT kernel that performs ray tracing based on the Nvidia sample code [10] using RT Cores. We implement $Kernel_{CD}$ to be the CD kernel that uses CUDA Cores, and we implement $Kernel_{TC}$ to be the TC kernel that uses Tensor Cores. $Kernel_{CD}$ and $Kernel_{TC}$ perform pure computation using registers and do not perform any memory operations. Each of these three kernels has the same solo-run computation time, and each occupies half of the memory resources on the SM.

$$Makespan\ Reduction = \frac{T_1 + T_2 - T_{colo}}{T_1 + T_2} \quad (1)$$

We use the metric *Makespan Reduction* to measure the parallelism between the two kernels. Equation 1 calculates the makespan when co-running two kernels. In this equation, T_1, T_2, and T_{colo} represent the solo-run time of the first

Table 3: The *Makespan Reduction* of six possible co-located kernel pairs.

Co-located kernel pair	Makespan Reduction
$Kernel_{RT} + Kernel_{CD}$	39.87%
$Kernel_{RT} + Kernel_{TC}$	34.62%
$Kernel_{CD} + Kernel_{TC}$	44.12%
$Kernel_{RT} + Kernel_{RT}$	0%
$Kernel_{CD} + Kernel_{CD}$	0%
$Kernel_{TC} + Kernel_{TC}$	0%

Table 4: The SM ratio required by cloud games on RTX 3090. The fps target is set to 120fps.

Game	High Quality	Medium Quality	Low Quality
Served Steel	100%	68.97%	23.44%
DUTM	100%	73.17%	47.62%
Quake 2	100%	72.29%	44.12%
Ascent	100%	39.73%	25.64%

kernel, the solo-run time of the second kernel, and the total makespan of completing the two kernels at co-location.

Table 3 shows the results of all kernel co-location pairs. When two $Kernel_{RT}$, two $Kernel_{CD}$ or two $Kernel_{TC}$ co-run, the makespan reduction is 0. When different kernels co-run, the makespan reduction is significant, with the minimum reduction being 34.62%. This is mainly because the two kernels run in parallel on the different cores on the SM. There is potential intra-SM parallelism if the co-running kernels use different processing units. Note that, the makespan reduction does not reach the theoretical value, which is 50%. This is because there is still implicit resource contention such as L1 cache and bus bandwidth contention.

2.3.2 Inter-SM sharing. Since a GPU contains multiple SMs, another intuitive method for application co-location is to partition SMs between different applications. If the cloud game could always occupy a certain amount of SM and leave the remaining SMs to the BE applications, there is no need to find a kernel to fit in the QoS headroom. This approach could solve the problem of under-utilized free GPU cycles.

Table 4 shows the SM ratio required by four cloud games, each with different rendering requirements. The SM ratio refers to the minimum SM ratio that supports the cloud game to meet the rendering QoS. As shown in the table, four cloud games require an average of 70% and a maximum of 90% SM in all cases. This implies that cloud games could occupy partial SMs to satisfy the QoS requirement.

2.4 Challenges

Although there is potential to utilize the intra-SM and inter-SM sharing to solve the problems of temporal sharing. There

are three main challenges lying behind exploiting these two-level sharing schemes:

❶ **There exists no straightforward method for fine-grained management of RT kernels.** RT kernels use a different programming language (e.g. Optix). The resource management like SM allocation goes beyond the scope of previous works. We have to find a way to control RT kernels in a fine-grained manner.

❷ **There is a large decision space involved in achieving the theoretical improvement of two-level SM sharing.** There are more resource contentions (shared memory, registers) among actual kernels and they can influence the actual gain of spatial sharing. We need to efficiently identify the optimal sharing scheme.

❸ **It is challenging to ensure each frame's QoS under varying rendering loads with spatial sharing enabled.** It is hard to preempt GPU kernels launched with spatial sharing, leading to more likely QoS co-location. When facing ever-changing rendering tasks, the scheduling policy needs to select the proper sharing scheme to avoid QoS violation.

3 OVERVIEW OF COMBO

To mitigate the above challenges, we propose COMBO, a high performance system that adaptively utilizes intra-SM and inter-SM sharing schemes to improve overall throughput while guaranteeing QoS for cloud games. COMBO is comprised of a *PTB adaptor*, a *duration predictor*, and a *sharing scheme selector*. The adaptor performs source-to-source compilation to convert kernels to resource-tunable versions based on Persistent-Thread-Block (PTB) method. COMBO exploits the converted kernels for fine-grained resource management. The predictor can precisely predict the duration of individual kernels and kernel co-locations with different sharing schemes. Based on the predicted rendering time and runtime workload, the scheme selector uses appropriate sharing schemes to launch BE kernels without QoS violation of the games' rendering.

In order to squeeze the GPU's idle resources, two sharing schemes are adopted in the scheme selector: *exact integrated-SM sharing*, and *related intra-SM sharing*. These two schemes are used to handle two load modes of cloud gaming respectively, which are stable load mode and dynamic load mode.

In the stable load mode, the rendering time of the frame is stable for a period of time. The scheme selector chooses the exact integrated SM sharing scheme for this load mode. Exact integrated SM sharing allocates the best-fit number of SMs for rendering kernels of cloud games and leaves the remaining SMs to the BE applications. The best-fit SM number is the minimum SMs that satisfy the frame's rendering QoS target, obtained from the duration predictor. Meantime,
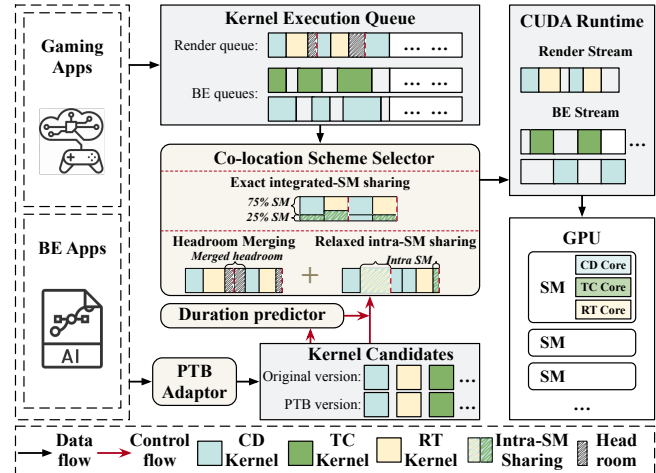

Figure 3: The overview of COMBO.

BE kernels could utilize the remaining SMs for intra-SM parallelism between CD and TC kernels.

In the dynamic load mode, the rendering time of the frame has a great variation. The scheme selector chooses the relaxed intra-SM sharing scheme for this load mode. Relaxed intra-SM sharing mainly utilizes the parallelism between RT kernels and CD kernels. While the co-running of RT kernels and CD kernels does not introduce QoS violation, COMBO co-locates the RT kernel and the CD kernel for intra-SM sharing. If the co-running of RT kernels and CD kernels cannot provide the performance gain, the relaxed intra-SM sharing falls back to the temporal sharing scheme.

4 RESOURCE MANAGEMENT FOR RT KERNELS

In this section, we first illustrate the necessity of resource management for intra-SM sharing. Although the inter-SM sharing requires adjusting the SM number for kernels, we do not illustrate its necessity due to its intuitiveness. Second, we propose the resource management method for RT kernels. Finally, we add the discussion about the co-location methods of different kernels.

4.1 Requirements for Intra-SM parallelism

We investigate whether real-world applications can benefit from the intra-SM parallelism. In this experiment, we choose 4 open-source RT kernels (*Pathtracer*, *cutouts*, *whitted*, and *motionblur*) from Nvidia Optix benchmark [10]. We select one high-performance TC kernel (*tzgemm*) from Nvidia [15] and use eight scientific application kernels from Parboil benchmark [30] suite as the CD kernel. For the co-location method, we choose CUDA stream to co-locate the RT kernel and other kernels.
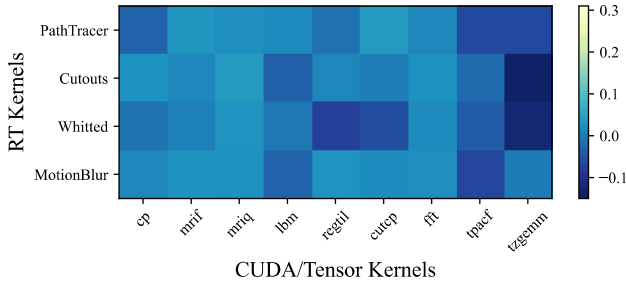
Figure 4: The makespan reduction of real-system applications under arbitrary intra-SM parallelism.

Figure 4 shows the makespan reduction of kernel pairs at co-location. All the kernels have the same solo-run duration. From the figure, the makespan reduction for most kernel pairs is around 0, indicating low parallel utilization of both types of computing cores. Meanwhile, a GPU kernel generally comprises a grid of blocks, with each block being scheduled onto a SM for computation. Therefore, no parallelism from the kernel co-location implies that co-located kernels' blocks could not reside on the SM simultaneously.

While there is potential parallelism between different computing cores, direct kernel co-running's inefficiency comes from the contention for resources on the SM. If one kernel occupies a large amount of explicit resources (e.g., thread slot and shared memory), another kernel cannot launch its thread block on the SM. Based on the above analysis, we could conclude that we need to solve the resource contention between kernels to enjoy the intra-SM parallelism. This necessitates resource management for RT kernels.

4.2 Resource management for RT kernel

Persistent-Thread-Block method. Since Nvidia GPU's grid scheduler launches kernels in an exhaustive manner [22], kernel's blocks are equally dispatched to all SMs immediately. While GPU kernels generally have a large number of blocks, they can easily consume up one type of resources on the SM. Thread slot and shared memory are the two most highly contested resources.

Many previous works focus on the resource management of CD kernels and TC kernels [24, 32, 39, 40]. They all adopt the Persistent-Thread-Block (PTB) method. PTB's idea is to treat each issued block as a persistent worker. With PTB, each persistent block is assigned some tasks that correspond to the original thread blocks. For each task, the persistent block needs to re-compute the block index to perform the computation correctly. A persistent thread block exits when it completes its assigned tasks. By converting the original kernel into its PTB version, the block number and the resource usage on each SM can be effectively controlled.

```
__global__ void render_ori(params){
    int idx_x = optixGetLaunchIndex().x;
    int idx_y = optixGetLaunchIndex().y;
    render_pixel(params, idx_x, idx_y)
}

#define blocksize 128
__global__ void render_ptb(params){
    int threadid = blockIdx.x * blocksize +
↪   threadIdx.x;
    for (int j = threadid; j < params.width *
↪   params.height; j += SM_NUM * blocksize){
        int idx_x = threadid % params.width;
        int idx_y = threadid / params.height;
        render_pixel(params, idx_x, idx_y)
    }
}
```

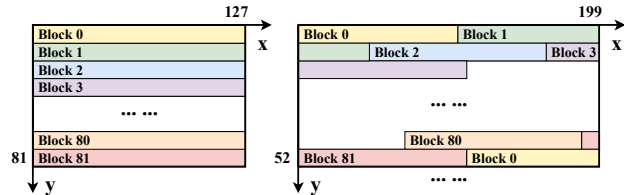**Figure 5: An example to construct PTB render kernel.**

Figure 6: Examples of rendering pixels with persistent thread blocks. The number of SMs is 82 here and each SM only possesses one persistent thread block. The x-axis represents the x coordinates of pixels and the y-axis represents the y coordinates of pixels.

However, the source-to-source compilation method for CD kernels and TC kernels cannot be applied to RT kernels. This is because RT kernels use a different programming language *Optix*. While CD kernels and TC kernels with CUDA rely on *blockIdx* and *threadIdx* to locate each thread, RT kernels use the *OptixGetLaunchIndex* to identify the pixel index. The *OptixGetLaunchIndex* represents the coordinate of a pixel in the rendered image. While previous works only deal with the block index re-computation, they cannot be directly applied to the RT kernel.

PTB adaption for RT kernels. Faced with this problem, we observe that the rendering task could also be treated as a CUDA-style kernel. We can re-introduce the notion of blocks and threads into the RT kernel by mapping the coordinate of a pixel to the thread index. Specifically, we could map each pixel-process task to the element-wise task, in which a pixel could map to a thread in the CUDA primitives. As shown in Figure 5, we could recalculate the pixel coordinate (idx_x, idx_y) with *blockIdx* and *threadIdx*.

Figure 5 presents the source-to-source transformation scheme for converting an RT kernel to the PTB version. After transforming the RT kernel to a PTB-based kernel version,
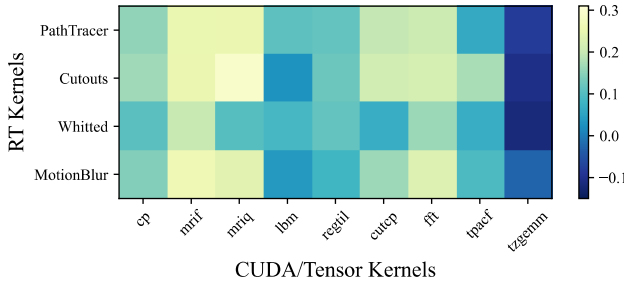
Figure 7: The makespan reduction of PTB real-system kernels under intra-SM parallelism.

we could adjust the persistent block number for RT kernel. The persistent thread block can accomplish the workloads originally executed by multiple blocks with the computation part wrapped in a for loop. Figure 6 shows how the rendering tasks of pixels are assigned to the persistent thread blocks.

With the transformed PTB-based RT kernel, we can re-conduct the co-location experiment. We locate the optimal persistent block number for each kernel using the method from previous works [39, 40]. The optimal persistent block number for one kernel is the minimum block number required to achieve the same performance as the original kernels. Experimental results show that all RT kernels' optimal persistent block number equals to the GPU's SM number.

Figure 7 shows the makespan reduction of kernel pairs at co-location after the PTB transformation. It can be observed that the makespan reduction between RT kernels and CD kernels is 13.7% on average, 25.9% at maximum. The kernel co-location with resource management could enjoy the intra-SM parallelism. However, when RT kernels are co-located with TC kernels, the makespan reduction is consistently 0 or even negative. This implies that RT kernels and TC kernels have severe implicit resource contention besides the thread slot and shared memory.

SM allocation for RT kernels. Furthermore, it is necessary to support RT kernels' computing on a partial number of SMs. Specifically, we adopt the method based on SM_{id} as previous works [24, 26, 32]. At the beginning, we launch the blocks to all the SMs. Inside the blocks, they will return immediately if the SM_{id} is not supposed to be used. Figure 8 shows the code transformation method to convert kernels into the SM-bounding version.

4.3 Discussions about co-location methods.

There are two primary co-location methods, which are kernel fusion and CUDA stream. While kernel fusion provides stable performance due to its deterministic block sequence, CUDA stream offers more flexibility without complicated

```
#define blocksize 128
__global__ void render_sm_bounding(params,
↪  sm_split_num){
    if (SM_id >= sm_split_num){
        return;
    }
    int threadid = blockIdx.x * blocksize +
↪  threadIdx.x;
    for (int j = threadid; j < params.width *
↪  params.height; j += sm_split_num * blocksize){
        int idx_x = threadid % params.width;
        int idx_y = threadid / params.height;
        render_pixel(params, idx_x, idx_y)
    }
}

__global__ void compute_sm_bounding(params,
↪  sm_split_num){
    if (SM_id < sm_split_num){
        return;
    }
    int block_id = SM_id - sm_split_num;
    int thread_id = threadIdx.x;
    for (;block_id < grid_dim; block_id +=
↪  (SM_NUM-sm_split_num)){
        compute_kernel(params, block_id, thread_id);
    }
}
```

Figure 8: An example to restrict SM usage of render kernel and compute kernel for inter-SM sharing.

compilation. In this section, we explain the reasons for not choosing kernel fusion and instead choosing CUDA stream.

Kernel fusion. Kernel fusion can be used to fuse multiple kernels that use different hardware resources (e.g., Tensor Cores and CUDA Cores) into a single kernel with nvcc [8] or nvrtc [11]. However, developers have to compile RT kernels with Optix [10] which has its own black-box compiler specifically designed for rendering shaders. After comprehensive analysis, we observe that Optix lacks the ability to explicitly access shared memory and cannot compile MMA operations to utilize Tensor Cores. Thus, RT kernels can neither be fused with CD kernels nor with TC kernels.

CUDA stream. By deploying workloads with different resource requirements into different CUDA streams, there is an opportunity to achieve parallel execution of tasks. As long as the GPU hardware resources (e.g., threads, shared memory) are not fully occupied and the remaining resources are sufficient to accommodate a kernel from another stream, that kernel can be launched and executed in parallel with existing kernels. Thus, the CUDA multi-stream mechanism allows both intra-SM parallelism and inter-SM parallelism.

With Optix retaining the concept of CUDA streams, we can further exploit spatial multi-tasking by launching kernels from different workloads onto separate CUDA streams. Specifically, we assign one stream to the render application, one stream to applications with CUDA kernels, and one stream to applications with tensor kernels.
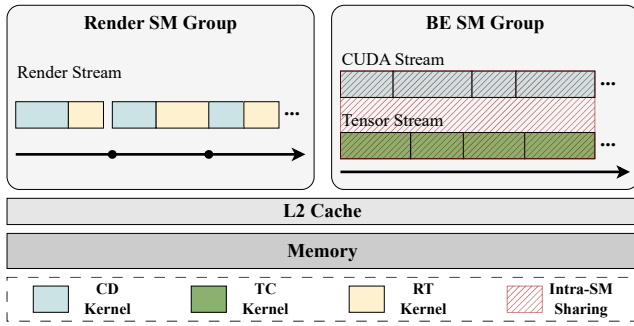
Figure 9: Timeline of exact integrated SM sharing.

5 SCHEDULING WITH TWO-LEVEL SM SHARING

In this section, we discuss how COMBO adopts an appropriate scheduling strategy to ensure QoS and improve GPU throughput when co-locating games and BE applications.

5.1 Constructing Sharing Schemes

In COMBO, the principle for constructing the sharing scheme is to maximize the throughput of BE applications, while ensuring the QoS of the games. To eliminate the QoS violation of rendering game frames, we can either precisely control the duration of rendering kernels through SM-level isolation, or subtly select kernels that would not violate the rendering application's QoS from BE applications. To maximize the throughput, we can either activate the idle computing units and squeeze the under-utilized free cycles. With the above principles in mind, COMBO first proposes two spatial sharing schemes. In the two schemes, three CUDA streams are created and sustained: one for running the games, one for running the CD kernels of the BE application, and the last one for running the TC kernels of the BE application.

5.1.1 Exact Integrated SM Sharing. In exact integrated SM sharing, the GPU is divided into two SM groups, where the render SM group runs the games and the BE SM group runs the BE applications, shown in Figure 9. We use an offline profiling method to determine the optimal SM allocation of two groups. To maximize the throughput of BE applications, we minimize the SM number allocated to the render SM group of the games while ensuring the QoS of each frame.

This co-location approach has a notable characteristic: as an inter-SM parallelism, the two groups do not share resources across the SM groups. Therefore, there is no shared memory or L1 cache contention between the games and the BE applications, with only L2 cache and DRAM contention. Meanwhile, as shown in Table 5, the L2 cache and DRAM throughput of rendering kernels in Optix is relatively low. This implies that the interference between the two groups is minimal during co-location. Thus, when determining the

Table 5: The L2 throughput of RT kernels.

	PathTracer	Cutouts	Whitted	MotionBlur
L2 throughput	7.92%	9.38%	4.81%	9.4%
DRAM throughput	9.47%	4.79%	6.51%	8.38%

proper SM allocation of the render SM group, we only need the duration predictor to predict the duration of rendering kernels, without considering the impact of the BE SM group.

Apart from the inter-SM sharing between the render and the BE SM group, there also exists intra-SM sharing inside the BE SM group. As depicted in the right of Figure 9, the CUDA Cores and Tensor Cores are both activated to maximize the BE application throughput.

5.1.2 Relaxed Intra-SM Sharing. The second scheduling scheme of COMBO is the relaxed intra-SM sharing. In this scheme, we first provision the whole GPU for running CD kernels of from games. Then we only allow the intra-SM sharing either for RT kernels and CD kernels from BE tasks, or CD kernels and TC kernels both from BE tasks. There are some reasons for resource contention for this design, especially in terms of the L1 cache and shared memory.

On the RTX3090, the total capacity of L1 cache and shared memory is 128KB. In the general compute mode, it is allocated as 64KB for L1 cache and 64KB for shared memory. However, during the execution of the graphics workload, it is allocated as 64KB for L1 cache, 48KB for shared memory, and 16KB for the graphics pipeline. If we want to co-run a compute kernel and an RT kernel, GPU must allocate L1 and shared memory according to the graphics workload mode. TC kernels for matrix multiplication typically require more than 40KB of shared memory, and RT kernels dynamically allocate at least 8KB of shared memory. This results in significant interference between TC kernels and RT kernels when they co-locate. In fact, according to Figure 7, the co-location of RT kernels and TC kernels leads to marginal or even negative makespan reduction, indicating that this co-location pair does not provide a performance improvement. CD kernels in games also consume more than 32KB shared memory. Therefore, they can not be co-located with TC kernels.

We can eliminate the aforementioned two pairs of co-location combinations from the combination candidates. This leaves us with two remaining co-location choices: one is to co-locate the RT kernel of the games with the CD kernel of the BE application, and the other is to co-locate the CD kernel with TC kernel both from the BE applications.

To pick proper kernels to co-run at runtime and ensure no QoS violation while rendering games' frames, we subtly choose kernels from the CUDA stream host queue and the Tensor stream host queue to launch. First, we would pop a CD kernel from the CUDA stream host queue and launch it onto the CUDA stream if it would not violate the QoS of the
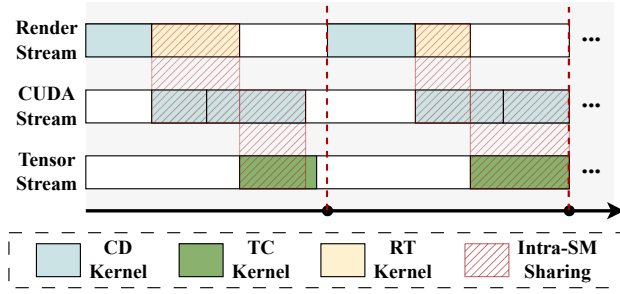
Figure 10: Timeline of relaxed intra-SM sharing.

games. Then, we select TC kernels from the Tensor stream host queue to co-locate them with the CD kernels in the CUDA stream as shown in Figure 10.

5.2 Comparing Two Spatial Sharing Schemes

In this subsection, we investigate performance of the two schemes with an experiment under varying rendering workload. We set the QoS target for the game to 16.7ms (60fps). In Experiment A, the workload of the games is constant, with a rendering time of 14.8ms per frame, and the BE kernel is a mriq kernel with a duration of 1.2ms. In Experiment B, the workload of the game varies. The rendering time fluctuates between 12ms and 16ms per frame while the average rendering time and the BE kernel duration are the same as Experiment A. In this dynamic workload scenario, the exact inter-SM strategy requires constantly changing the SM group allocation to meet the QoS target of the game. We use an offline profiling method to determine the optimal allocation scheme. However, each time the allocation scheme is changed, there is a sacrifice of the BE application in that frame to meet the QoS target. This issue will be further explained in detail in §5.3.

Under the above experimental configurations, the throughput of the two schemes is shown in Table 6. It can be observed that exact integrated SM strategy has a significantly higher throughput than relaxed intra-SM strategy in Experiment A. This is because in the relaxed intra-SM strategy, the QoS headroom is small, and running the BE kernel twice per frame would lead to QoS violations. However, sending only one BE application kernel per frame would waste approximately 0.6ms of free GPU cycles. In the exact inter-SM scheme, when the compute kernel is sent to the compute SM group, it has no effect on the rendering kernels, allowing the compute SM group to remain occupied continuously, thereby eliminating GPU free cycles.

In Experiment B, the relaxed intra-SM strategy performs better. This is because under dynamic workload, the SM group allocation scheme changes approximately every three

Table 6: The number of executed mriq kernels per frame under exact integrated SM sharing scheme and relaxed intra-SM sharing scheme.

	stable workload	dynamic workload
exact integrated SM	1.45	1
relaxed intra-SM	0.998	1.16

frames. Each time a change occurs, the BE task is blocked during the frame, resulting in a decrease in throughput.

From these two experiments, we can conclude that the exact integrated SM strategy is superior under a static rendering workload, while the relaxed intra-SM strategy is more suitable for a dynamic rendering workload.

5.3 Scheme Switching Strategy

In Section 5.2, we discover that the proposed two schedule schemes are suitable for different rendering workloads. The scheme switching logic is: the exact integrated SM strategy is selected when the rendering workload is static, while the relaxed intra-SM strategy is selected for a dynamic rendering workload. The workload state is judged by Equation 3.

$$state = \frac{max(rendertime) - min(rendertime)}{avg(rendertime)} \quad (2)$$

$$wokload\ state\ is \begin{cases} static, & state < 10\% \\ dynamic, & state \geq 10\% \end{cases} \quad (3)$$

If the parameter *state* is 9%, the workload is judged as static and the exact integrated SM strategy is employed, but the SM allocation scheme still may be adjusted. With the adjusting comes a problem that can violate the QoS target.

For the sake of simplicity in our discussion, let's assume that one SM group runs only the RT kernel while the other group runs only the TC kernel. As is shown in Figure 11, we have a GPU with 82 SMs in total. In the first frame, the rendering workload requires 50 SMs allocated to the game to meet the QoS target. In the second frame, the rendering workload changes, requiring 60 SMs. Ideally, as shown in Figure 11(a), both the TC kernel and the RT kernel finish simultaneously in each frame and everything goes well. However, in practice, as depicted in Figure 11(b), the TC kernel may finish later than the RT kernel. At the end of the first frame, the TC kernel still occupies SMs 51 to 82. This leads to increased latency for the blocks running on SMs 51 to 62, ultimately violating the QoS target for that frame. To eliminate this impact, we propose an optimization strategy, as illustrated in Figure 11(c). In the second frame, the original version of the RT kernel is launched. Unlike the SM-bound version, the original version does not bind each block to a specific SM. Instead, it launches blocks on any available SM. This eliminates the issue of increased latency in the SM-bound version of the kernel.
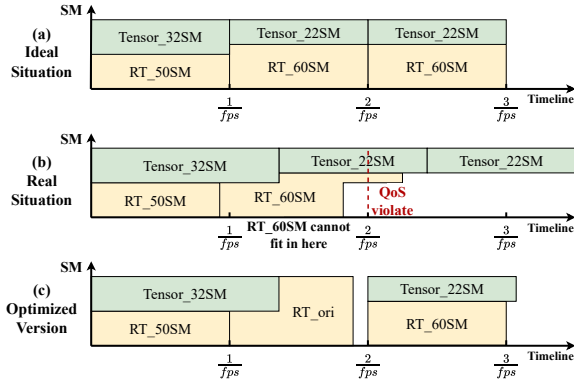
Figure 11: Timeline of kernels when SM group allocation adjusts in 3 situations. This GPU has 82 SMs.

When the workload of the rendering task transitions from static to dynamic, the strategy switches from integrated SM to relaxed intra-SM. In the relaxed intra-SM strategy, the rt kernel occupies all the SMs, but compute SM group may still occupy some SMs. This also can lead to an increase in rendering latency. In this case, we need to send an original version of the kernel during the frame of the scheme switch.

5.4 Partitioning SM Group

As to the exact integrated SM sharing strategy, the partitioning of SMs has a significant impact on both QoS and throughput. To achieve the QoS guarantee for the games, more SMs should be allocated to it. Conversely, to improve the GPU throughput, more SMs should be allocated to the BE application. Therefore, the optimal allocation strategy is to allocate just exactly the required number of SMs to the game, enabling it to meet the QoS objectives precisely. This necessitates the prediction of the execution time for the game's rendering kernels.

Rendering kernels are consisted of CD kernels and RT kernels. The execution time prediction formula is shown in Equation 4, where $height \times width$ represents the size of the render frame, i.e., the number of pixels, $sample$ denotes the number of ray samples per unit area, $num_{triangle}$ denotes the number of triangles in the scene, and a and b are parameters related to the render kernels. This method exhibits high accuracy in predicting rendering time, with a maximum error not exceeding 5.16%.

$$
\begin{aligned}
rendertime_{estimate} &= time_{rt} + time_{cuda} \\
time_{rt} &= a \times height \times width \times sample \\
time_{cuda} &= b \times height \times width \times num_{triangle}
\end{aligned}
\tag{4}
$$

With the observation of low interference and rt kernel prediction model, we can now decide the number of SMs allocated to the rendering kernels. The execution time of the render kernel is inversely proportional to the number of
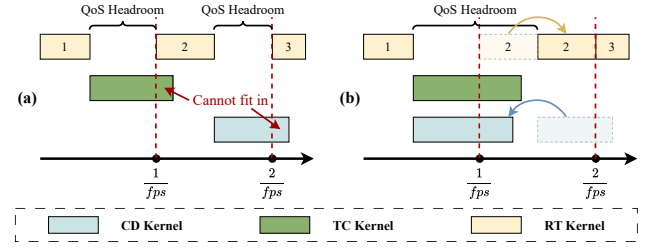
SMs used. Therefore, we can use Equation 5 to determine the number of SMs allocated to the rendering kernels. SM_{render} is the number of SMs allocated to rendering kernels. All the information needed can be collected on the host side and SM_{render} can be calculated before kernel launch.

$$
\begin{aligned}
SM_{render} &= SM_{total} \times \frac{QoS\ target}{rendertime_{estimate}} \\
&= SM_{total} \times \frac{1}{fps \times rendertime_{estimate}} \\
SM_{compute} &= SM_{total} - SM_{render}
\end{aligned}
\tag{5}
$$

Figure 12: The timeline of 3 kernels before and after headroom merge.

5.5 Merging Adjacent Headrooms

Under the relaxed intra-SM scheme, QoS headroom still exists. If the headroom is too small it can lead to a decrease in BE task throughput. Combo proposes a headroom merge method to combine the headroom of consecutive frames into a larger headroom. The specific approach is to shift the start time of the second frame's rendering ($\frac{1}{fps} - rendertime$) units later. A larger headroom means more potential parallel opportunities. As shown in Figure 1, the QoS headroom is enough for the CD kernel to fit in, but too small for both TC kernel and CD kernel to fit in. Due to the continuity of the rendering workload, the next few frame's rendering time usually does not change significantly. Therefore, the TC kernel cannot be launched for several consecutive frames. In such cases, if a headroom merge is performed, both CD and TC kernel can be fitted into the larger headroom without affecting QoS. This is not the only case that can fit in more kernels. Figure 12 shows more cases.

6 EVALUATION

In this section, we describe the implementation of Combo and reveal its performance improvement in cloud gaming.

6.1 Implementation of Combo

We implement Combo on top of the CUDA runtime and Combo supports graphics applications using Optix. There are other graphics libraries that support hardware-accelerated

pipeline tracing, such as Vulkan[16], and DirectX[4]. How-ever, the gap between them and CUDA is too big. They lack software concepts such as CUDA context and CUDA stream. In contrast, Optix, developed by NVIDIA, is based on CUDA. It not only preserves software concepts like CUDA context and CUDA stream but also follows the CUDA programming model. Given that Optix is easily integrated with CUDA programs, it is suitable for implementation in this work.

To evaluate Combo, we implement the PTB adaptor as a source-to-source compiler. This compiler takes the kernels from Optix rendering applications and BE applications as input. Firstly, it transforms these kernels into the PTB mode and then converts them into the SM-bounding mode using the transformation method introduced in §4.2. Next, we in-troduce the duration predictor, which is constructed offline. It co-locates the RT kernels, CUDA kernels, and Tensor ker-nels in a pair-wise mode for all possible combinations and builds a performance surface. The above process is similar to previous work[39].

6.2 Experimental Setup

Table 7 shows the detailed experimental setup. We choose four mainstream cloud games (*Servered Steel*, *Deliver Us to The Moon*, *Quake* 2, *The Ascent*). We use the training tasks of four commonly used DNN models (*Resnet*50, *VGG*16, *Densenet*, *Inception*) and eight scientific applications from Parboil as BE applications. The cloud games contain RT ker-nels and CD kernels. The DNN training tasks contain CD kernels and TC kernels. The scientific applications only con-tain the CD kernel. The BE application's kernels are captured in a FIFO queue and then launched in order.

Since the cloud gaming corporations do not open source their codes, we simulate the games with the RT kernels (*pathtracer*, *cutouts*, *whitted*, and *motionblur*) from the NVIDIA Optix SDK [10]. Specifically, we simulate the imple-mentation of cloud games based on the kernel distribution from the industry [2, 3, 12, 14]. Besides, we collect the game trace from real game scenarios, which comprises the stable load mode and the dynamic load mode. Table 2 in §2 lists the detailed configurations of the cloud games.

The experiments are carried out on a server equipped with an Nvidia RTX 3090 GPU. Combo does not rely on any particular hardware features of 3090 and is easy to set up on other gaming-oriented GPUs that both integrate RT Cores and CUDA Cores.

6.3 Improving Throughput

In this subsection, we compare Combo with Pilotfish, a tem-poral sharing scheme that improves GPU utilization while ensuring QoS. Since no applications contain three types of computing cores, we need to co-locate the cloud games with

Table 7: Hardware and software specifications.

	Specification
CPU	Intel(R) Xeon(R) W-2223 CPU @ 3.60GHz
GPU	Nvidia GeForce RTX 3090
Software	CUDA Version 11.1.96, Optix SDK 7.1.0
Cloud games	Servered Steel, Deliver Us to The Moon, Quake 2, The Ascent
BE tasks	cp, mrif, mriq, lbm, regtil, cutcp, fft, tpacf, Resnet50, VGG16, Densenet, Inception

two BE applications. These two BE applications are the DNN training task (TC kernels and CD kernels) and the scientific application (Only CD kernels).

Equation 6 calculates the throughput improvement of Combo compared with Pilotfish. Since different applications have different durations, we choose to count the duration of different kernels. In this equation, T_{combo_tc} and T_{combo_cd} represent the overall durations for TC kernels and CD ker-nels under Combo, respectively. $T_{pilotfish\_tc}$ and $T_{pilotfish\_cd}$ represent the overall durations for TC kernels and CD ker-nels under Pilotfish, respectively.

$$Throughput\ improve = \frac{T_{combo\_tc} + T_{combo\_cd}}{T_{pilotfish\_tc} + T_{pilotfish\_cd}} \quad (6)$$

Figure 13 compares the throughput of the BE applications when adopting Combo and Pilotfish. From the figure, Combo achieves an average of 14.0% (and up to 38.2%) improvement over Pilotfish. Combo improves the throughput for all 128 (=4×8x4) co-location groups because it exploits the intra-SM and inter-SM sharing on the GPU. Specifically, Combo could adaptively switch the sharing scheme for different load modes. When the cloud game exhibits a stable load mode, Combo could utilize the exact inter-SM sharing to harvest all the free GPU cycles. When the cloud game exhibits a dynamic load mode, Combo could utilize the relaxed intra-SM sharing and headroom merge to host as many CD kernels as possible. As a comparison, Pilotfish only relies on the temporal sharing scheme to utilize the limited free GPU cycles.

We also observe that Combo achieves the highest through-put improvements with the game *Quake* 2. The throughput improvement with *Quake* 2 is 16.8% on average, while others are 12.9%, 8.55%, and 12.6%. This is because the performance gain from intra-SM parallelism depends on the duration of RT kernels. While *Quake* 2 has the RT kernels accounting for 50.6% overall duration, its improvements is the highest.

Moreover, we observe that Combo achieves higher through-put improvements for compute-intensive BE applications (*cp*, *mrif*, *mriq*, *fft*). This is because memory-intensive ap-plications (*lbm*, *regti*, *cutcp*, *tpacf*) require more memory resources and their co-runs face more resource contention,
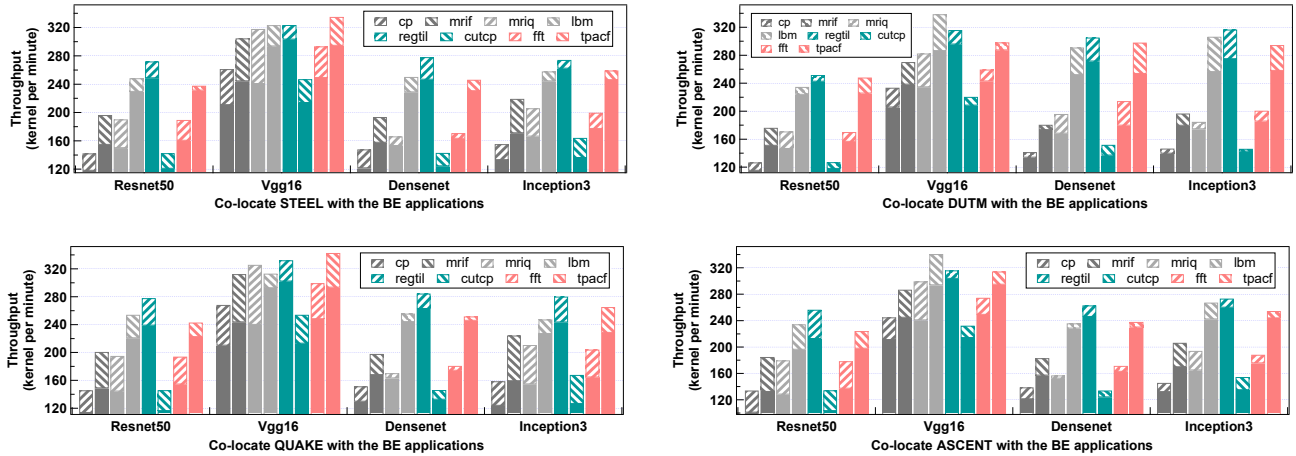
Figure 13: The throughput comparison of BE applications between Combo and PilotFish. Solid columns stand for PilotFish, dashed columns stand for Combo.
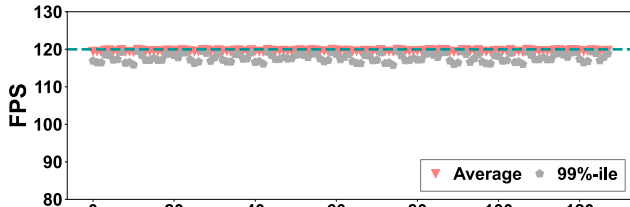

Figure 14: The average and 99%-ile FPS of the rendering application in all the 128 co-location groups with Combo.
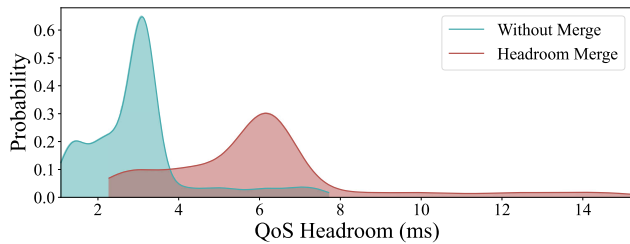

Figure 15: The QoS headroom distribution.

leading to lower throughput improvements. These findings are consistent with the experimental results in §4.2.

6.4 Guaranteeing QoS

Figure 14 shows the 99%-ile and average FPS for all the co-location groups. It can be observed that the average FPS under Combo is higher than 119, and the 99%-ile FPS is higher than 117. This implies that Combo achieves almost the same 99%-ile FPS compared to that without co-location. The great QoS performance of Combo comes from its highest priority for QoS guarantee.

In the exact integrated-SM scheme, the BE applications and the cloud game are executed in different SM groups.

Combo first allocates the best-fit SM resources to the cloud game, which could satisfy its QoS requirement. Second, these two groups have limited interference with each other. Therefore, the cloud game hardly has QoS violations.

While the cloud game enters the dynamic load mode, the scheduling scheme switches to relaxed intra-SM sharing. This is to avoid potential increases in rendering latency that may occur in the exact integrated-SM scheme. In the dynamic load mode, Combo co-locates the RT kernel with CD kernel only if it would not introduce the QoS violations.

At the switching point of the two sharing schemes, Combo directly uses the original kernels to serve the rendering of the next frame. In this way, Combo could guarantee that the sharing scheme switching does not introduce the QoS violation. Benefiting from the above designs, Combo achieves great QoS performance.

6.5 Effectiveness of Headroom Merge

Figure 16 shows the throughput improvement of Combo compared with Combo-NM. Combo-NM refers to the system that disables the headroom merge. We also utilize the Equation 6 for computation. As observed from the figure, Combo achieves an average of 6.74% (and up to 19.8%) improvement over Combo-NM. This performance gains attributes to the enlarged QoS headroom.

As shown in Figure 15, the headroom merge significantly increases the QoS headroom. This implies two points. Firstly, the headroom merge provides greater flexibility in co-locating the RT kernel with CD kernels. More options from BE kernels could be considered for higher system throughput. Secondly, some BE applications may have been blocked without the
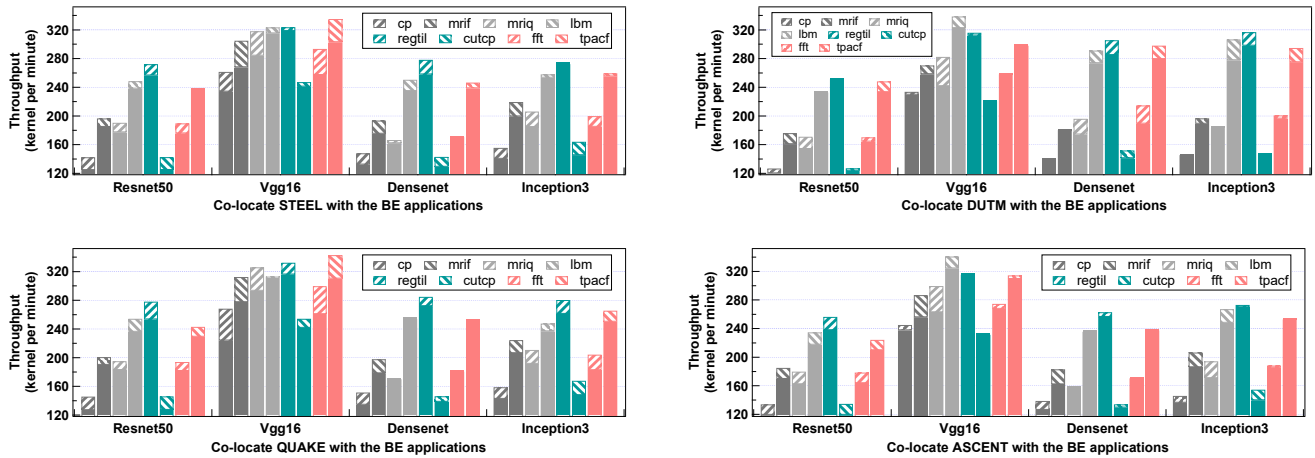
Figure 16: The throughput comparison of BE applications between using headroom merge and no merge. Solid columns stand for Combo-NM, dashed columns stand for Combo.
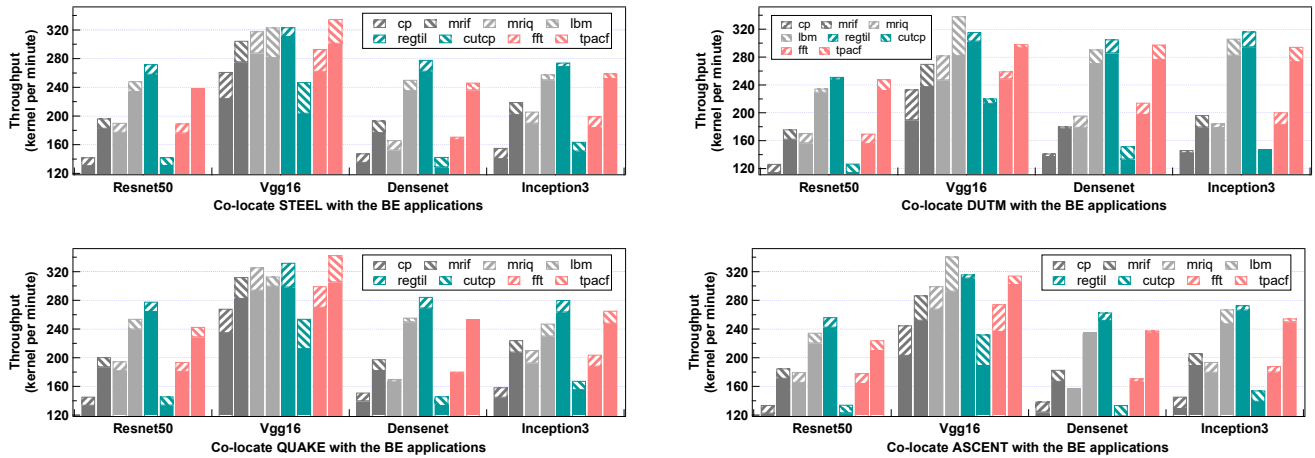

Figure 17: The throughput comparison of BE applications between Combo and relaxed intra-SM sharing only. Solid columns stand for intra-SM sharing scheme, dashed columns stand for Combo.
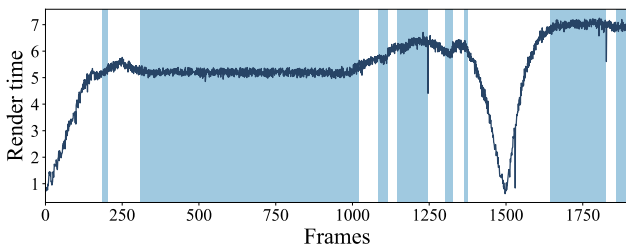

Figure 18: The trace of the rendering time of frames in THI. In the white region, the rendering time of the frame fluctuates and thus relaxed intra-SM scheme is applied. In the blue region, the rendering time of the frame is stable and thus the scheduling scheme is switched to the exact integrated SM sharing scheme.

headroom merge for their kernels cannot fit in the QoS headroom. With the headroom merge, these applications have the opportunity to continue the computation.

6.6 Effectiveness of Scheme Switch

Combo can adaptively switch the sharing scheme when the load mode of cloud game changes. As shown in Figure 18, the rendering time of the frame varies in a real-world cloud game *Servered Steel*. Combo can switch between two schemes according to the cloud game's load mode. The load mode could be perceived by predicting the frame's rendering time.

We compare the throughput of BE applications between using Combo and exploiting relaxed intra-SM sharing scheme without switching to exact integrated-SM sharing scheme.

Figure 17 shows the throughput improvement of combo compared with intra-SM sharing scheme. Combo achieves an average of 8.43% (and up to 23.3%) improvement. Besides, the throughput improvement is more significant on VGG16, whose kernel duration is longer. As the longer duration means less opportunity to fit in the headroom, it is hard for the kernels of VGG16 to fit in the QoS headroom. However, with exact integrated-SM sharing, it is always possible for VGG16 to execute on the separated SM group.

6.7 Overhead.

In this subsection we add discussion about the overhead introduced by Combo.

Offline overhead. The offline overhead of Combo primarily consists of two parts. The first part involves the transformation of kernels into their PTB and SM-bounding versions for all applications. In our experiments, we have 4 gaming applications, 4 DNN applications, and 8 CUDA applications. This process takes less than 1s. The second part of the offline overhead is the construction of the duration predictor. It takes 0.583s on average to build the prediction model.

Online overhead. During the runtime of Combo, the overhead includes the prediction latency, the latency for determining the scheme, and the kernel launch latency. The three components involved in the latency are executed sequentially, so their latencies cannot overlap. The average prediction time is 0.06ms, the selection of the scheme takes 0.02ms, and the latency for kernel launch is less than 0.1ms.

7 RELATED WORK

There are several prior works that focus on improving GPU utilization with time-division multiplexing. TimeGraph [25] schedules GPU commands using an event-driven model and introduces two priority-based scheduling strategies to ensure the performance of high-priority kernels. Baymax [17] increases GPU utilization by optimizing user queries through kernel reordering. AntMan [33] optimizes deep learning training by accommodating fluctuating resource demands and utilizing spare GPU resources to co-execute multiple jobs. However, these works cannot handle constantly changing game rendering tasks.

Space-division multiplexing including inter-SM sharing and intra-SM sharing is another promising approach to improve GPU utilization [28, 31, 34, 38–40]. Representing GPU inter-SM sharing, Nvidia's MPS [7] allows multiple applications to concurrently share a GPU with static partitioning. However, its inability to manage dynamic cloud gaming loads limits its broad applicability and effectiveness. There are also other works that realize inter-SM spatial multitasking through code transformation [24, 32] and extra hardware

design [41, 42], yet the potential of application-level throughput improvement is not revealed in these works. With inter-SM sharing, Laius [38] maximizes the throughput of batch applications while ensuring the QoS of user-facing services. Some other works aim to improve the throughput of DNN inference systems [18–21, 35]. In terms of intra-SM sharing, Plasticine [40] introduces a system of persistent and elastic blocks to alleviate resource contention, enhancing GPU throughput by co-locating TC and CD kernels. Warped-Slicer [34] utilizes a dynamic intra-SM slicing strategy to partition SM resources across different kernels and it considers the interference effect of shared resource usage.

Focusing on the domain of cloud gaming, numerous works have been proposed to enhance the utilization of various resources under game co-location scenarios. In some previous works, multiple games are co-located together to improve resource utilization [27, 29, 36]. Vgris [29] virtualizes the GPU and provides different partitions for multiple games. vGASA [36] employs an adaptive scheduling approach to handle multiple games in a cloud gaming scenario, ensuring service level agreement compliance. Besides co-location among games, PilotFish [37] co-locates cloud games with deep learning training jobs and enhances GPU utilization by harvesting idle GPU cycles. It strategically schedules and preempts kernels based on anticipated idle periods. Besides cloud gaming, RT cores on GPU can also be leveraged in other scientific problems. RTNN [43] utilizes RT cores to solve the neighbor searching problem. It utilizes query scheduling and partitioning to tackle the issue of hardware under-utilization in this scenario.

8 CONCLUSION

In this paper, we propose Combo to maximize the utilization of GPUs in the cloud gaming scenario. Combo leverages both intra-SM sharing and inter-SM sharing to improve the throughput of BE applications, while guaranteeing the QoS of rendering games' frames. We devise a compilation method to utilize RT cores for fine-grained resource management, as well as two efficient spatial sharing schemes considering runtime rendering load. Experimental results show that Combo can achieve up to 38.2% throughput improvement compared with the state-of-the-art solutions.

ACKNOWLEDGMENTS

REFERENCES

[1] 2023. Amazon AppStream 2.0. https://aws.amazon.com/appstream2/. Accessed: 2023-06-08.

[2] 2023. The Ascent. https://curvegames.com/our-games/the-ascent/. Accessed: 2023-06-08.

[3] 2023. Deliver Us to The Moon. https://www.deliverusthemoon.com/. Accessed: 2023-06-08.

[4] 2023. DirectX. https://learn.microsoft.com/en-us/windows/win32/directx-sdk--august-2009-. Accessed: 2023-06-10.

[5] 2023. GeForce RTX 3090 Family. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090-3090ti/. Accessed: 2023-06-08.

[6] 2023. Microsoft's Xbox Remote Play. https://www.xbox.com/en-US/consoles/remote-play. Accessed: 2023-06-08.

[7] 2023. NVIDIA Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html. Accessed: 2023-06-08.

[8] 2023. NVIDIA CUDA Compiler Driver NVCC. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. Accessed: 2023-06-08.

[9] 2023. Nvidia Geforce Now. https://www.nvidia.com/en-us/geforce-now/. Accessed: 2023-06-08.

[10] 2023. NVIDIA OptiX Ray Tracing Engine. https://developer.nvidia.com/rtx/ray-tracing/optix. Accessed: 2023-06-08.

[11] 2023. NVRTC. https://docs.nvidia.com/cuda/nvrtc/index.html. Accessed: 2023-06-08.

[12] 2023. Quake 2 RTX. https://www.nvidia.com/en-us/geforce/campaigns/quake-II-rtx/. Accessed: 2023-06-08.

[13] 2023. RTX Technology. https://developer.nvidia.com/rtx/ray-tracing. Accessed: 2023-06-08.

[14] 2023. Served Steel. https://digerati.games/game/severed-steel/. Accessed: 2023-06-08.

[15] 2023. tensor core example code. https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/cudaTensorCoreGemm. Accessed: 2023-06-08.

[16] 2023. Vulkan. https://www.vulkan.org/. Accessed: 2023-06-10.

[17] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. *SIGPLAN Not.* 51, 4 (mar 2016), 681–696. https://doi.org/10.1145/2954679.2872368

[18] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on {Multi-GPU} Servers with {Spatio-Temporal} Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 199–216.

[19] Marcus Chow, Ali Jahanshahi, and Daniel Wong. 2023. KRISP: Enabling Kernel-wise RIght-sizing for Spatial Partitioned GPU Inference Servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 624–637.

[20] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. 2021. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[21] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. 2020. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 492–506.

[22] Guin Gilman, Samuel S Ogden, Tian Guo, and Robert J Walls. 2021. Demystifying the placement policies of the NVIDIA GPU thread block scheduler for concurrent kernels. *ACM SIGMETRICS Performance Evaluation Review* 48, 3 (2021), 81–88.

[23] Kshitij Gupta, Jeff A Stuart, and John D Owens. 2012. *A study of persistent threads style GPU programming for GPGPU workloads.* IEEE.

[24] Saksham Jain, Iljoo Baek, Shige Wang, and Ragunathan Rajkumar. 2019. Fractional GPUs: Software-based compute and memory bandwidth reservation for GPUs. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 29–41.

[25] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, Yutaka Ishikawa, et al. 2011. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*. 17–30.

[26] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeongon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *2014 IEEE 20th international symposium on high performance computer architecture (HPCA)*. IEEE, 260–271.

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

[28] Jason Jong Kyu Park, Yongjun Park, and Scott Mahlke. 2017. Dynamic resource management for efficient utilization of multitasking gpus. In *Proceedings of the twenty-second international conference on architectural support for programming languages and operating systems*. 527–540.

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

[30] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.

[31] Zhenning Wang, Jun Yang, Rami Melhem, Bruce Childers, Youtao Zhang, and Minyi Guo. 2017. Quality of service support for fine-grained sharing on GPUs. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 269–281.

[32] Bo Wu, Guoyang Chen, Dong Li, Xipeng Shen, and Jeffrey Vetter. 2015. Enabling and exploiting flexible task assignment on GPU through SM-centric program transformations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 119–130.

[33] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning.. In *OSDI*. 533–548.

[34] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. 2016. Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 230–242. https://doi.org/10.1109/ISCA.2016.29

[35] Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. 2021. Automated runtime-aware scheduling for multi-tenant dnn inference on gpu. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

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

[37] Wei Zhang, Binghao Chen, Zhenhua Han, Quan Chen, Peng Cheng, Fan Yang, Ran Shu, Yuqing Yang, and Minyi Guo. 2022. {PilotFish}: Harvesting Free Cycles of Cloud Gaming with Deep Learning Training. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 217–232.

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

[39] Han Zhao, Weihao Cui, Quan Chen, Youtao Zhang, Yanchao Lu, Chao Li, Jingwen Leng, and Minyi Guo. 2022. Tacker: Tensor-cuda core kernel fusion for improving the gpu utilization while ensuring qos. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 800–813.

[40] Han Zhao, Weihao Cui, Quan Chen, Jieru Zhao, Jingwen Leng, and Minyi Guo. 2021. Exploiting Intra-SM Parallelism in GPUs via Persistent and Elastic Blocks. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*. 290–298. https://doi.org/10.1109/ICCD53106.2021.00054

[41] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 653–663.

[42] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. Hsm: A hybrid slowdown model for multitasking gpus. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems*. 1371–1385.

[43] Yuhao Zhu. 2022. RTNN: accelerating neighbor search using hardware ray tracing. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 76–89.