Gemini: Enabling Multi-Tenant GPU Sharing Based on Kernel Burst Estimation

Hung-Hsin Chen, En-Te Lin, Yu-Min Chou, and Jerry Chou, *Member, IEEE*

Abstract—Recent years have seen rapid adoption of GPUs in various types of platforms because of the tremendous throughput powered by massive parallelism. However, as the computing power of GPU continues to grow at a rapid pace, it also becomes harder to utilize these additional resources effectively with the support of GPU sharing. In this work, we designed and implemented *Gemini*, a user-space runtime scheduling framework to enable fine-grained GPU allocation control with support for multi-tenancy and elastic allocation, which are critical for cloud and resource providers. Our key idea is to introduce the concept of *kernel burst*, which refers to a group of consecutive kernels launched together without being interrupted by synchronous events. Based on the characteristics of kernel burst, we proposed a low overhead *event-driven monitor* and a *dynamic time-sharing scheduler* to achieve our goals. Our experiment evaluations using five types of GPU applications show that Gemini enabled multi-tenant and elastic GPU allocation with less than 5% performance overhead. Furthermore, compared to static scheduling, Gemini achieved 20%~30% performance improvement without requiring prior knowledge of applications.

Index Terms—GPU, multi-tenancy, resource allocation, performance, scheduling

1 Introduction

Recent years have seen rapid adoption of GPUs in various types of platforms because of the tremendous throughput powered by massive parallelism. The use of GPUs has emerged as the means for achieving extreme-scale, cost-effective, and high-performance computing. To expose the performance and improve the programmability of GPUs for general-purpose processing, software developers and software engineers often have to rely on a GPU software stack like CUDA. CUDA is a parallel computing platform providing API libraries, compiler, runtime, and GPU drivers for NVIDIA GPU accelerators. CUDA's programming model and libraries let applications offload compute-intensive workloads as *kernels* to the GPU for acceleration. A kernel consists of a group of thread blocks that can execute independently and in parallel to utilize GPU resources. Kernels are managed by the hardware scheduler and executed non-preemptive and asynchronously to the CPU host in order to maximize GPU throughput. However, as the computing power of GPU continues to grow rapidly, it also becomes increasingly difficult to utilize these additional resources effectively in reality. Due to synchronization and load imbalanced problems, many scientific (MPI-based) codes have to be carefully tuned and optimized to achieve better parallel efficiency [2]. [23] also reported 20–70% GPU utilization from running Parboil2 benchmark suite. The utilization problem is expected to worsen with the increasing performance gap between GPU and CPU/IO and the growing diversity of applications ported to GPUs.

Enabling GPU sharing among computing tasks is one of the effective solutions to address the GPU utilization problem in terms of both computing and memory resources. However, GPUs have been designed to maximize the performance of a single application and thus assume exclusive access from a single process. Although recent solutions from NVIDIA (e.g., Multi-Process Service (MPS) [22], Hyper-Q [21], and STREAM) allows multiple processes to access a shared GPU and execute multiple kernels concurrently, there is still limited control provided to operating systems or software. Hence, it has drawn noticeable attention from operating systems [10], [14], [15], [20], [23], [26], [27] and real-time [13], [25], [35] research communities to develop multi-tasking control on shared GPU for improved efficiency and fairness. However, the proprietary GPU stack and the asynchronous, non-preemptive nature of the kernel make it difficult to obtain the necessary information and control to manage GPU from the host side. As a result, most solutions are based on custom API library [14], driver [8], [15], [26] or OS module [20], [27]. Others need compile-time optimization or profiling [10], [23], [31].

This work aims to achieve fine-grained resource management control on shared GPU with GPU stack transparency and without prior knowledge from applications. Our implementation, called *Gemini*, is implemented as a lightweight runtime library that exists between the CUDA API interface and applications for monitoring and controlling kernel submissions to GPUs so that the GPU can be shared among applications in a time slicing manner. To enable fine-grained control with low overhead, we introduce the concept of *kernel burst*, which refers to a group of consecutive kernels launched together without being interrupted by synchronous events. Based on kernel burst, we design an *event-driven monitoring* technique in user space to capture and measure runtime

• *The authors are with the Department of Computer Science, National Tsing Hua University, Hsinchu 300044, Taiwan. E-mail: jim90247@gapp.nthu.edu.tw, {etlin, ymchou, jchou}@lsalab.cs.nthu.edu.tw.*

Manuscript received 29 May 2021; revised 3 October 2021; accepted 5 October 2021. Date of publication 11 October 2021; date of current version 8 March 2023. (Corresponding author: Jerry Chou.)
Recommended for acceptance by J. Zhai.
Digital Object Identifier no. 10.1109/TCC.2021.3119205

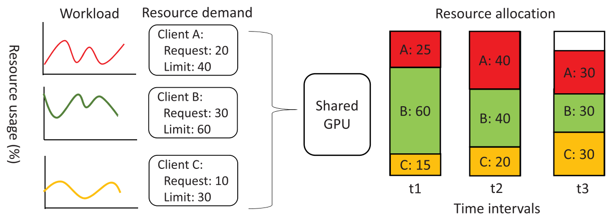


Fig. 1. Example of multi-tenant and elastic resource allocation. The resource allocation can be elastically adjusted among clients under resource demand constraints.

kernel execution behavior without introducing synchronization points between GPU and CPU. A *token-based time-sharing scheduler* is implemented to coordinate GPU usage among clients/tenants to provide a multi-tenant and elastic allocation model. We also proposed a dynamic quota strategy and token revocation scheme based on the estimated kernel burst time to adapt dynamic workload patterns. Our experimental evaluations using five different types of GPU workloads show that Gemini achieved fine-grained resource management and guaranteed resource allocation requirements with less than 5% performance overhead. Furthermore, Gemini could adapt to workload patterns at runtime without additional parameter tuning or application profiling. Comparing to static scheduling, Gemini achieved 20%∼30% performance improvement and ensured GPU resources could be fully utilized under resource requirement constraints. Although our work is based on NVIDIA GPUs and CUDA software stack, our solution can be easily applied to other similar GPU programming libraries and software stack, like OpenCL.

The rest of the paper is structured as follows. Section 2 describes the objectives and challenges of GPU sharing. Section 3 details the design and implementation of Gemini. The experimental results are shown in Section 4. Section 5 discusses related work. This paper concludes in Section 6.

2 Problem Descriptions

2.1 Objective

The utilization of a GPU is defined as the percentage of time one or more GPU kernels are running over a period of time. The goal of our GPU sharing is to maximize GPU utilization by supporting *elastic* and *multi-tenant* resource sharing, which is a common resource allocation model needed by cloud or cluster managers. In this model, a client is a tenant that contains an application process or container instance. Each client (i.e., tenant) is associated with a specified resource demand between a minimum value (*request*) and a maximum value (*limit*) according to their workload variations. The demand represents the percentage of time a client can perform kernel execution on a GPU. Hence, scheduling the computing tasks of multiple clients (tenants) to the same GPU can maximize GPU utilization. However, clients' tenancy must be satisfied by ensuring the actual GPU usage of a client is within its resource demand specification.

As illustrated by Fig. 1, *Multi-tenancy* ensures that a client is guaranteed to be allocated as much GPU as it requests, and it cannot use more GPU than its *limit*. *Elastic allocation* allows the residual capacity to be elastically allocated among clients as long as the allocations do not exceed clients' resource limits so that the overall GPU utilization for computing can be maximized. Therefore, as shown in Fig. 1, the actual resource allocation of clients keeps changing over time due to workload variations. But the allocation never violates the *request* and *limit* specified by the clients. As a result, GPUs can be shared among clients with guaranteed allocations while maximizing utilization.

2.2 Challenges

Enabling fine-grained and controlled GPU sharing is a challenging problem because of the two main reasons below.

Black-Box GPU Stack. The application interface for GPUs (including libraries, runtime systems, drivers, hardware) is usually inaccessible to programmers. Hence, only limited GPU execution information and control are available. Some existing solutions (e.g., GDev [15], GPUvm [30], TimeGraph [14]) modify the vendor-provided software stacks to schedule GPU kernels. But it is difficult to ensure their compatibility and support for new GPU architectures. Therefore, the right level of the interface must be chosen to gain enough GPU control with high applicability to diverse applications.

Asynchronous and Non-Preemptive Kernel: To maximize throughput, GPU and CPU mainly cooperate asynchronously. Unless explicit synchronization functions are called between CPU and GPU, software programs cannot obtain the start time and end time of kernel execution. But these synchronization calls lead to significant drops in GPU throughput at the same time. Furthermore, modern GPU architecture doesn't support kernel preemption. That means once a kernel is launched on GPU, it is managed by a hardware scheduler, and its execution cannot be stopped or suspended by OS. Therefore, GPU control must be done proactively rather than reactively. Also, synchronization calls must be avoided to minimize performance degradation.

2.3 Design Requirements

Multi-Tenant Isolation. Our GPU sharing mechanism is to enable multi-tenant GPU from a cloud/resource provider perspective. So our first priority is to isolate the GPU kernel execution time among clients and guarantee their resource usages are controlled under a resource demand capacity. Like most cloud shared resource environments [9], we allow users to specify the maximum(i.e., limit) and minimum(i.e., request) resource usage of a job. So the resource manager can maximize resource utilization under elastic usage demands.

High Utilization. Resource isolation and utilization often conflict with each other. Especially the asynchronous GPU kernel execution behavior makes it challenging to track accurate GPU usage status. Hence our goal is to maximize the GPU utilization without violating the resource isolation constraints.

Low Overhead. Kernel executions typically have microsecond-level latencies. Therefore, our GPU sharing approach is strongly motivated to minimize overhead on a per-kernel or per-request basis. At the same time, the GPU throughput should be maintained while providing fine-grained resource control.

Workload Adaptation. Application workload can significantly impact resource management decisions, and the workload characteristics can vary widely between applications.

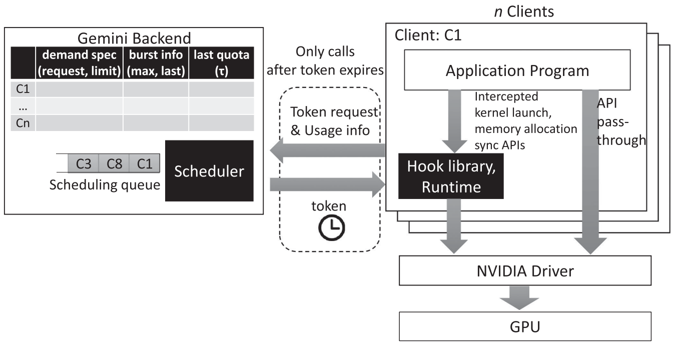


Fig. 2. The system architecture of Gemini.

Hence, we designed an adaptive resource management algorithm and technique to achieve optimized performance without any fine-tuned parameter setting.

Software Transparency. Many previous GPU sharing approaches rely on offline profiling [10], compiler [23], [31], customized API [27], or customized OS module [20]. But we want our approach to be transparent to GPU applications for better compatibility with the existing software stack.

3 Gemini

3.1 Approach Overview

To meet the requirements and overcome the challenges discussed in Section 2, we designed and implemented a runtime GPU sharing framework, called *Gemini*. The resource isolation is ensured by enabling a time-sharing solution based on a *token-based scheduling architecture* (described in Section 3.2), and proposing a *token revocation scheme* (described in Section 3.5.3) to avoid non-preemptive kernels from exceeding their scheduling time quota. In addition, we ensure high resource utilization under isolated GPU usage by supporting *elastic allocation* (described in Section 3.5.1) for users.

The key innovation of our approach is to explore the behavior of *kernel burst* (described in Section 3.3) from GPU programs so that a fine-grained and low overhead *event-driven monitor* (described in Section 3.4) can be developed for tracking asynchronous and non-preemptive kernel executions. Taking advantage of the runtime monitor information, we developed a *dynamic quota scheduler* (described in Section 3.5.2) without the need for parameter tuning. Furthermore, we support GPU *sharing with memory over-subscription* through CUDA Unified Memory technique and propose strategies to minimize the memory transfer delay of unified memory (described in Section 3.6). Our solution is implemented as a lightweight *CUDA API hook* module (a Linux shared library described in Section 3.2), and our code is open source available on github [6].

3.2 System Design

The system architecture of Gemini is shown in Fig. 2. To ensure our approach is compatible with existing GPU software stack and transparent to user applications, Gemini consists of two components: a frontend CUDA API hook runtime library inserted in the client application environment and a centralized backend scheduler for coordinating GPU usage among clients.

The API hook is a Linux shared library whose path is added to the `LD_PRELOAD` Linux environment variable. The `LD_PRELOAD` trick supported by Linux OS [1] is a useful technique to influence the linkage of shared libraries and the resolution of symbols (functions) at runtime. In particular, before any other library, the dynamic loader will first load the shared libraries in `LD_PRELOAD`. Therefore, we can re-implement the function of the intercepted library call by executing our inserting code before calling the original implementation. In our work, we use this trick to allows our hook library to intercept CUDA API calls from applications and retrieve runtime information or even override the original CUDA API.

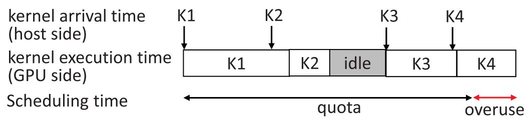
In our work, we need to intercept three kinds of CUDA API calls: kernel launch, memory management, and host-side synchronization. As described in Section 3.4, the APIs of kernel launch and host-side synchronization are intercepted to capture the client's GPU usage pattern and request tokens from the scheduler for computing resources. The memory management APIs are intercepted to support memory over-subscription through the CUDA Unified Memory technique as described in Section 3.6. To minimize runtime overhead, we carefully design and implement the hook library so that it only performs scheduling coordination and bookkeeping when the client application voluntarily calls CUDA synchronization APIs. The CUDA API calls that are not intercepted by our hook library are directly sent to the CUDA driver. Therefore, the overhead can be hidden and overlapped by the synchronization events without causing performance or GPU throughput degradation.

The backend scheduler implements a token-based scheduling framework that ensures GPU resource is allocated to clients in a time-sharing manner. A token represents the timeslice, and the quota of a token is the length of a timeslice. Thus, only the API hooks with valid tokens can pass kernel launch calls from clients to the GPU driver for execution. A token becomes invalid when its quota is expired, and its hook library must request a new token from the scheduler for future kernel execution. Therefore, the backend scheduler can isolate GPU usage between clients and control client's resource allocations by scheduling tokens among clients according to their resource demand settings of request and limit.

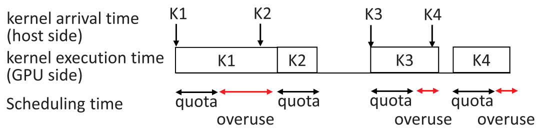
The setting of the token quota is critical to the scheduling performance. As illustrated by the example in Fig. 3, if the token quota is too long, there could be more client idle time included in the timeslice, which can lower the GPU utilization because the token blocks other clients from submitting kernels to GPU. On the other hand, if the token quota is too short, clients must request valid tokens from the scheduler more frequently, which causes higher synchronization overhead between API hook and scheduler. The chance of overuse also increases when the token quota becomes shorter. Hence, finding the best token quota is not a trivial question, and it is highly dependent on the application behaviors.

3.3 Kernel Burst

A "*kernel burst*" refers to a group of consecutive kernels launched together without being interrupted by synchronous


kernel arrival time (host side)
K1
K2
K3
K4
kernel execution time (GPU side)
K1
K2
idle
K3
K4
Scheduling time
quota
overuse

(a) A larger quota causes more GPU idle time.



(b) A smaller quota causes more scheduling overhead and resource overuse.

Fig. 3. An illustration of the resource overuse and quota selection problems. K1∼K4 denotes kernel events.

events. Kernel bursts can be commonly observed from GPU applications because of the typical GPU programming model below:

1) copy data to GPU device memory
2) launch CUDA kernels without data dependency
3) wait for kernels to complete
4) copy results back to CPU host memory

Although these GPU commands can be issued asynchronously from a process (or a CUDA stream), their corresponding events are still executed sequentially on GPU, as shown in Fig. 4. Hence, the GPU usage patterns can be modeled by a sequence of interleaving phases between execution and synchronization. The execution phase contains the timeframe with kernel executing on GPU, and the synchronization phase contains the timeframe without any kernel execution. Hence, the synchronization phase includes the time for handling the synchronization events within a GPU device or between CPU and GPU, as well as the pure GPU idle time without receiving any GPU activity from the host. Scheduling tokens and allocating resources with respect to kernel burst instead of individual kernel launch has the following advantages.

1) The coordination between clients and scheduler within a burst period should be avoided because it may cause unnecessary scheduling overhead and additional GPU idle time when all other clients don't require GPU.
2) The overhead from tracking the kernel time can be significantly reduced, as described in Section 3.4.
3) Kernel burst consists of a group of kernels, so the execution time variation between kernel bursts should be less than the variation between individual kernels. Therefore, the stability of kernel burst can help us to make more accurate time predictions as described in Section 3.5.
4) We still have the capability to manage resources in a finer time granularity by setting the token quota as a fraction of kernel burst time.

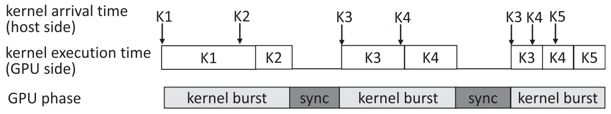


Fig. 4. An illustration of kernel burst and synchronization phase on GPU. K1∼K5 denotes kernel events.

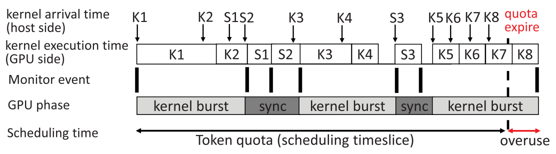


Fig. 5. An illustration of the event-driven monitor. Monitor events are piggyback on synchronization events from client applications and scheduling events to determine kernel burst time and overuse. K1∼K8 denotes kernel events, and S1∼S3 denotes synchronization events.

3.4 Event-Driven Monitoring

Our monitoring mechanism is implemented in the API hook. Therefore, the monitor can only tap into the CUDA APIs and get triggered when these APIs are called by applications. The goal of the monitor is to identify kernel bursts from applications and correctly record their actual start time and end time for execution. The main challenge is that CUDA events are recorded asynchronously to the CPU. So we have to insert synchronization events (such as `cudaEventSynchronize`) in order to let GPU notify our monitor when a kernel burst starts and completes. But the questions are *"when to insert these synchronization monitoring events to GPU?"*, and *"how to receive the notifications from these synchronization events without blocking the applications that have been monitored?"*.

To avoid the synchronization overhead of monitoring, we piggyback our monitoring events when applications call synchronization events themselves. This is sufficient to detect kernel burst because we assume kernel burst is a sequence of kernel launch events followed by a synchronization event. As long as we can insert a monitor event right before users' synchronization event, the completion time of the previous kernel burst can be captured. The only exception is when the completion time of a kernel burst exceeds its token quota. In this case, the execution of a kernel burst could be forced to stop by an invalid token instead of a synchronization event. As a result, we schedule a monitor event when the token is expired so that the completion time of the last kernel burst can be reported. This monitor event doesn't cause additional overhead because, in this situation, the API hook has to block applications and request a new token from the scheduler anyway. More importantly, it allows us to measure the amount of overuse in each scheduling timeslice and ensure our monitored resource allocation matches the actual resource usage.

On the other hand, the start of a kernel burst can only occur in two possible cases. One is the time when the API hook receives a token from the scheduler to begin submitting the waiting kernels to GPUs, and the other one is the time when the first kernel launch command is called after a synchronization event. To detect these two time points, we only need to add the monitor events after completing the user synchronization events or API hook scheduling events. Again, our monitor events can be piggybacked with existing synchronization points.

As illustrated by the example in Fig. 5, we use K1∼K8 to denote the kernel events and use S1∼S8 to denote the synchronization event. Our monitor only schedules events in the following situations:

- Before & after the completion of synchronization events.

- When the current token is expired.
- When a new token is received from the scheduler.
- When the first kernel launch API is called after a synchronization event.

Clearly, these synchronization points are sufficient to accurately identify and measure the execution time of each kernel burst. The only exception is that the GPU idle time between a kernel event and its followed synchronization event will be counted into kernel burst time (i.e., like the idle time between K4 and S3). But this time should be bounded and limited in practice because synchronization events are often scheduled right after kernel events. Our early token revocation scheme will detect this situation and return the token to the scheduler to maximize GPU utilization.

Finally, to receive the notification of our monitor events without blocking applications, our API hook spawns a set of monitor threads during program initiation. `cudaEventRecord` is called on the main thread (i.e., GPU stream) to schedule synchronization points, and `cudaEventSynchronize` is called on the monitoring threads to record kernel burst time. Since the GPU phases interleaved between execution and synchronization, no more than a few monitoring threads need to be maintained in our implementation.

3.5 Dynamic & Elastic Scheduling

Our backend scheduler has two main goals. One is to support elastic allocation and maximize GPU utilization under the resource demand constraints from clients. The other one is to provide dynamic quota scheduling so that the proper token quota can be selected corresponding to the workload patterns of individual clients. We detail our elastic allocation and dynamic quota scheduling strategy implemented in the scheduler below. Finally, we propose a token revocation scheme to permit clients to adjust scheduling quota proactively in order to prevent GPU idle and increase GPU utilization.

3.5.1 Elastic Allocation

To allocate GPU resources in a time-sharing manner, our scheduler follows the four steps below to decide which client should receive the next valid token from the scheduling queue. These steps are taken by the scheduler periodically when the previous token is expired with clients waiting in the scheduling queue or when a new client request arrives without any client holding the valid token.

Step1. Calculate Allocations. We use a sliding window to measure the current resource allocation amount of each client. As illustrated by the example in Fig. 6, the allocation amount of a client is its accumulated timeslice intervals within the sliding window. There will be no overlap between client's timeslices under our time-sharing allocation model, except in the occurrence of overuse. To ensure the total resource allocation amount is equal to or less than 100% utilization, we assume all the clients with an overlapped timeslice receive an equal share of the allocation amount.

Step2. Filter Requests. To enforce the limit constraint in resource demand, the requests from clients whose allocations already exceed resource limits will not be selected.

Step3. Prioritize Requests. From the rest of the valid client requests in the scheduling queue, we first choose the client whose allocation is farthest from its resource request. If all clients have reached their resource request, we next select the client whose allocation is farthest from its resource limit. In other words, our first priority is to satisfy the minimum resource requirement from clients. If there is residual capacity on GPUs, we will continue to allocate the resource to the clients until they reach their maximum resource requirement.

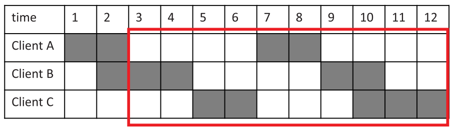


Fig. 6. An illustration of how GPU allocation is calculated by the scheduler. Given a sliding window size of 10 time intervals, the allocations of each client at time interval 12 are A (20%), B (35%), and C (45%), respectively.

Step4. Determine Token Quota. Finally, we must decide the quota of a token. As discussed in Section 3.2, the size of the quota presents a trade-off between scheduling overhead and GPU utilization. Hence, we proposed to use an estimated kernel burst time as the quota as explained next.

3.5.2 Dynamic Quota

As explained in Section 3.3, an ideal quota setting is to use the expected execution time of the kernel burst that issues token requests. But the execution time cannot be known in advance, and the workload pattern can vary over time. Therefore, our approach lets the API hook provide some statistics of the kernel burst of its client to the scheduler. Then the scheduler uses a smooth function to gradually adjust the token quota of a client according to its estimated burst time from the client. We detail how quota is decided in below.

Let $\{KB_1, \ldots, KB_n\}$ be the set of execution time measured from the last n kernel bust events by our event-driven monitor. We first attempt to merge kernel bursts separated by short synchronization intervals. Two thresholds decide the definition of short. One is the minimum communication delay between client and scheduler, denoted by γ. The other is the maximum tolerance ratio for GPU idle, denoted by ϵ. $\epsilon = 5\%$ means we can tolerate at most 5% GPU idle time from a client. Accordingly, let I_i be the synchronization time intervals between KB_i and KB_{i+1}. We merge the execution time of KB_i and KB_{i+1}, if $I_i < \gamma$ or $I_i/KB_{avg} < \epsilon$, where KB_{avg} is the average time of $KB_1 \sim KB_n$.

Next, the API hook reports the values of KB_{max} and KB_{last} in its token request sent to the scheduler, where KB_{max} is the 90 percentile values of kernel burst time from the merged kernel bursts, and KB_{last} is the time of the last merged kernel burst.

Then the scheduler estimates the next kernel burst time, KB_{est}, as

$$KB_{est} = \alpha \times KB_{max} + (1 - \alpha) \times KB_{last}, \tag{1}$$

where α is a parameter ratio between 0 and 1. The intuition of the above equation is that the estimated burst time should be bounded between the maximum and previous burst time.

Finally, the token quota, τ, is decided by

$$\tau = \beta \times \tau_{last} + (1 - \beta) \times KB_{est}, \tag{2}$$

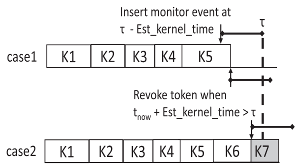


Fig. 7. An illustration of the two conditions for token revocation.

where β is another parameter ratio between 0 and 1, and τ_{last} is the last token quota received by the client. The intuition of the above equation is that the quota should be gradually moved toward the estimated kernel burst time. Noted, the setting of α and β only affects the aggressiveness of our adaption. The quota is still learned from the workload patterns, not decided by these parameters.

3.5.3 *Token Revocation*

The token quota is decided by the scheduler based on statistical estimation, and a longer token quota is preferred in order to reduce scheduling coordination and avoid resource overuse (i.e., as indicated by (1)). So the actual kernel burst time can still be different from the given token quota. To further adjust the token quota according to the client's runtime behavior, we allow a client to revoke its token before quota expiration when the client finds the remaining quota time is not enough to execute the next kernel. Based on estimated kernel execution time, token revocation can occur in the two situations illustrated in Fig. 7. The first case occurs when the remaining quota time at the end of a kernel execution is less than the estimated kernel execution time. The second case occurs when a kernel launch call arrives with the remaining quota time is less than the estimated kernel time. In the first case, we have to insert a monitor event when the remaining quota time is less than the estimated kernel time so that we can be notified at the end of the last kernel execution and revoke the token immediately to avoid GPU idle or overuse. In the second case, we only need to check the remaining quota time when kernel calls arrive and block the kernel if we need to. Noted, our monitor can only measure the time of each kernel burst, not the individual kernel. Therefore, we roughly estimate the kernel time by dividing the kernel burst time by the number of kernel calls that occur within a burst period. In general, the token revocation scheme helps us to further reduce the problem of resource overuse and idle from the client side.

3.6 Memory Sharing and Optimization

While the computing resource is shared in the time domain, the memory is shared in the space domain in our approach. In our tenancy model, a client must also specify its maximum memory usage in the resource specification, and its memory usage should never exceed its requested limit throughout its execution. The memory limit information allows the cluster manager to avoid memory over-subscription or control the level of memory contention when dispatching clients to GPUs. Therefore, our GPU sharing library must also guarantee the memory usage constraint at runtime.

Similar to our resource throttling technique for computing, *Gemini* intercepts all the memory allocation API calls to the CUDA driver in order to track the actual memory usage from each client. As shown in Table 1, CUDA provides several different kinds of API calls for users to allocate and free memory, including `cuMemAllocXXX` calls, and array allocation calls, such as `cuArrayCreate`, `cuMipmappedArrayCreate`. Hence, all of these memory allocation calls are intercepted by our hook library to ensure the correct usage accounting. We also found that some applications, like Tensorflow, attempt to allocate all the memory space on GPU at the initial time to manage the memory usage by themselves. To ensure these applications can also work properly under limited memory usage, we also intercept `cuDeviceTotalMem` and `cuMemGetInfo` to override their returned device memory size with the maximum memory usage size requested by users. Thus, the memory size viewed by applications can be consistent with the memory size requested in their resource specifications. When a client attempts to use more memory space than its limit, our CUDA API hook library at the client side directly denies the memory allocation API by throwing an out-of-memory exception to the user programs. Hence, clients basically run their GPU program on a virtual GPU with limited computing capacity and memory space.

TABLE 1
Hooked CUDA Driver APIs

CUDA Driver API
Memory Allocation/Free `cuMemAlloc` `cuMemAllocManaged` `cuMemAllocPitch` `cuMemFree`
Array Create/Free `cuArrayCreate` `cuArray3DCreate` `cuMipmappedArrayCreate` `cuArrayDestroy` `cuMipmappedArrayDestroy`
Memory Information `cuDeviceTotalMem` `cuMemGetInfo`
Kernel Launch `cuLaunchKernel` `cuLaunchCooperativeKernel`
Host-side Synchronization `cuMemcpyAtoH` `cuMemcpyDtoH` `cuMemcpyHtoA` `cuMemcpyHtoD` `cuCtxSynchronize`

To limit the amount of GPU memory allocation from a client is relatively straightforward. But the limited size of physical memory on GPUs could become an obstacle for GPU sharing when memory over-subscription is not allowed. This is because some GPU applications may require a huge amount of GPU memory space but have relatively low GPU utilization. As a result, memory usage may limit the number of concurrent clients on a shared GPU. To address this problem, we propose to use the unified memory technique. Unified memory was first introduced in CUDA 6.0 release for NVIDIA Kepler GPU architecture to provide a single, unified virtual address space to applications for accessing CPU and GPU memory. The managed memory is shared between CPU and GPU, and the CUDA driver migrates data between CPU and GPU when needed automatically and transparently to applications. Therefore, through unified memory, we let GPUs use the host memory as a swap space for the GPU memory content and only keep the active memory in GPU so that the GPU could be shared by more applications. To achieve this, *Gemini* simply replaces all the intercepted memory and array allocation/free APIs with the `cudaMallocManaged` unified memory allocation API. Since the data migration of unified memory is transparent to the user programs, our approach will not affect user program behaviors.

However, unified memory could introduce a long memory access delay during CUDA kernel execution when its accessed data must be transferred from the host. The memory transfer delay can cause two issues. First, it causes a lower GPU utilization because the GPU becomes idle during transfer time. Second, it unexpectedly prolongs kernel execution time which can overuse problems. However, the overuse problem is not a concern under low GPU utilization because the overuse computing time of a kernel can be overlapped with the memory transfer time of another kernel. As a result, in the memory oversubscribed situation, the computing resource overuse behavior will not degrade performance or affect fairness (or isolation). Instead, it can help improve resource utilization in this situation. Therefore, the primary concern of memory over-subscription is to reduce memory transfer time and improve GPU utilization. To minimize the performance degradation from memory transfer delay, we consider two optimization strategies. One is to prefetch the memory content using the `cudaMemPrefetchAsync` API when the launched kernel is queued at the *Gemini* backend scheduler. Because the prefetch operation is implemented by a separate CUDA stream, we can overlap the kernel execution time and the data migration time among different clients to hide the memory access latency. The other strategy is to reduce the amount of data that needs to be prefetched by guessing the memory contents that are required for the kernel launch or by deallocating the unused memory contents from applications. As shown in a recent study [34], many applications, like deep learning, have periodic and regular memory access workload, and the majority of the memory allocation can be freed (e.g., tensors created for forward propagation) after an execution stage for reducing their memory footprint. Our evaluation in Section 4.4 shows these optimization strategies can effectively improve the performance by up to 40%.

4 Evaluations

4.1 Experiment Setup

We extensively evaluate our approach based on the implemented code made available on github [6]. We conducted experiments on a server with NVIDIA Tesla V100 GPUs. We chose five applications to evaluate the performance of our GPU sharing mechanism. Two of them are high-performance scientific simulations. Another two of them are GPU benchmark programs, and the last one is a deep learning inference server.

- NAMD is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems.

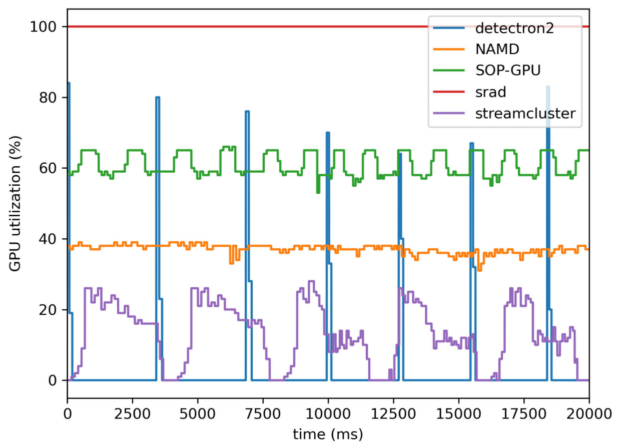


Fig. 8. The GPU utilization and usage pattern of our testing applications.

- SOP-GPU package is a scientific software package designed to perform Langevin Dynamics Simulations of the mechanical or thermal unfolding.
- srad and streamcluster are chosen from Rodinia benchmark [4], which is a benchmark suite for heterogeneous computing, such as GPU.
- detectron2 [32] is a PyTorch-based deep learning application developed by Facebook for object detection. Detectron2 originates from maskrcnn-benchmark [18], and it supports several deep learning models. Six models are used in our evaluation, and "Faster R_CNN X101-FPN" is the default choice if we don't specify. We deploy its inference server for the COCO2017 dataset [17] as our testing application.

As shown in Fig. 8, the GPU usage pattern and utilization of these applications are quite different. For instance, srad and NAMD have constant GPU usages over time, while SOP-GPU, streamcluster, detectron2 have periodic GPU usage patterns. Especially for detectron2, its periodic usage pattern and workload intensity (i.e., average GPU utilization) are determined by the inter-arrival time of inference requests ingested in the experiments.

In the experiments, we evaluate the performance of these applications under varied resource allocation scenarios. Hence, we show their baseline performances and resource usages as below. Fig. 9 plots their job execution time varied resource allocation. As can be observed, scientific simulations (NAMD and SOP-GPU) are more sensitive to resource amounts, while deep learning inference processing, like detectron2, can be less sensitive. For detectron2, its inference request processing time and GPU usage depend on the deep learning models. Fig. 10 shows the GPU usage under varied request inter-arrival time. As observed, most applications can reach near-optimal performance with less than 100% resource allocation. For instance, all our tested applications, except NAMD, can reach 90% of the peak performance with only 70% of the resource. Therefore, GPU sharing can definitely improve resource utilization and system throughput, and greater improvements can be expected with higher computing power GPUs.

In the rest of the section, we will present our evaluation results in three aspects. The first is to show our approach can enable GPU sharing with resource control for isolation and multi-tenancy. The second is to show the performance

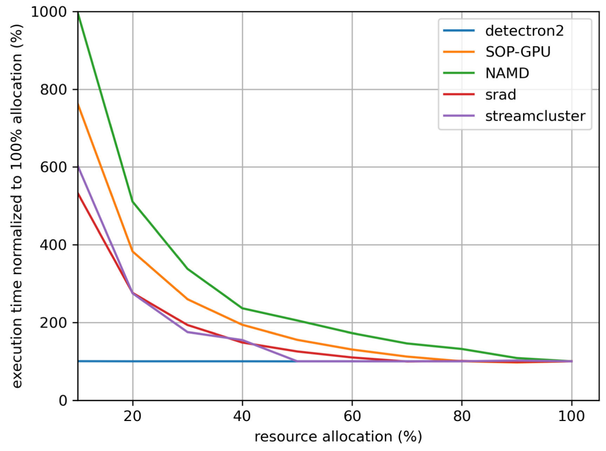


Fig. 9. The correlation between the application performance and resource allocation amount.

improvement of the dynamic quota over the static quota and analyze our kernel burst estimation technique. Last is to evaluate the overhead of our approach under different running environments, including standalone, concurrent, and memory over-subscription. Noted, the static quota solution can be considered as the previous work from GaiaGPU [11], which only proposed the scheduling framework and API interception technique without discussing or solving the memory allocation and fairness problem from asynchronous kernel execution.

4.2 Resource Sharing and Control Improvements

Our first set of experiments shows that our approach can improve GPU performance with resource sharing while providing controls for resource isolation and elastic allocation.

To show the sharing benefit, we consider the execution time of a workload consisting of the five testing application with equal running time for each of them, and no resource limit is given to these applications. We compare the execution time of the workload when the applications run concurrently on a shared GPU over the time when applications run sequentially on a non-shared GPU. The execution time of each application on a shared GPU is plotted in Fig. 11, where the execution time is normalized to the result from running the workload sequentially on a non-shared GPU. The total execution time of the workload on shared GPU is bounded by the slowest execution of an application. Hence,

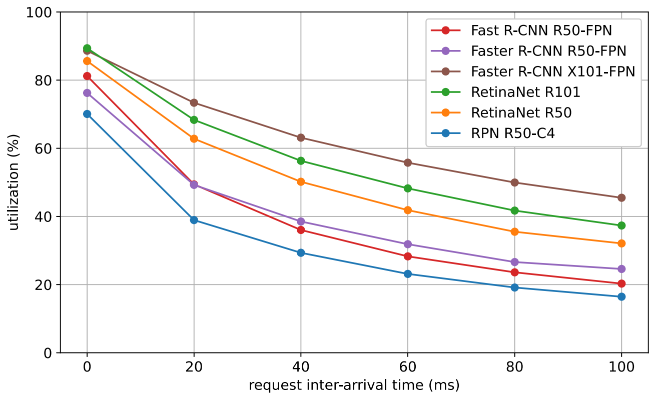


Fig. 10. The correlation between the GPU utilization and the request inter-arrival time of detectron2.

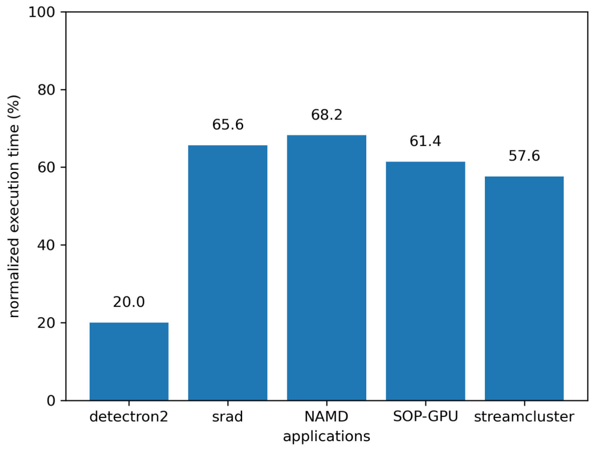


Fig. 11. The benefits of GPU sharing. Running a workload consisting of five applications on a shred GPU concurrently takes less than 70% of the time of running them sequentially on a non-shared GPU.

as observed, the total execution of the workload on a shared GPU takes only 70% of the time on a non-shared GPU. We can also find that the execution time of applications is varied, with more computing-intensive applications likely suffering from longer execution time. It also implies that applications on shared GPUs may obtain a different amount of resources and results in different performance impacts. Therefore, controlling resource allocation among applications on a shared GPU is critical to resource providers or system administrators for ensuring multi-tenancy and fairness, especially in multi-users or cloud environments.

To prove Gemini can allocate resources according to the user requests, we now run the workload with five applications on a shared GPU with a resource limit given to each application. We repeat the experiments 20 times, and the resource limits given to applications are randomly chosen between $0 \sim 100$ in each experimental run. If the applications actually receive the resource requested by the limit, its execution time should be the same as a result reported in Fig. 9 when the applications are running alone on a GPU. Hence, we refer to the results in Fig. 9 as the ideal performance and define the *normalized performance deviation* of an application in an experimental run as the difference of execution time between the observed value and its ideal performance divided by the ideal performance. Fig. 12 plots the cumulative distribution function (CDF) of all the normalized performance deviation values collected from all the applications and experimental runs (i.e., a total of 100 sampling points). As observed, more than 80% of normalized performance deviations are under 0.5, and less than 3% of the values are higher than 0.7. Therefore, Gemini effectively avoids the performance interference problem to deliver similar and consistent application performance even with coexisting jobs running on a shared GPU.

Finally, we show that Gemini can provide elastic allocation to maximize resource utilization while ensuring resource limit constraints. To demonstrate the elasticity, we dynamically add and remove the applications running on a shared GPU in order to observe how the resource will be redistributed. We use the workload of detectron2 with different deep learning models in this evaluation because the inference server can be started and stopped at any time.

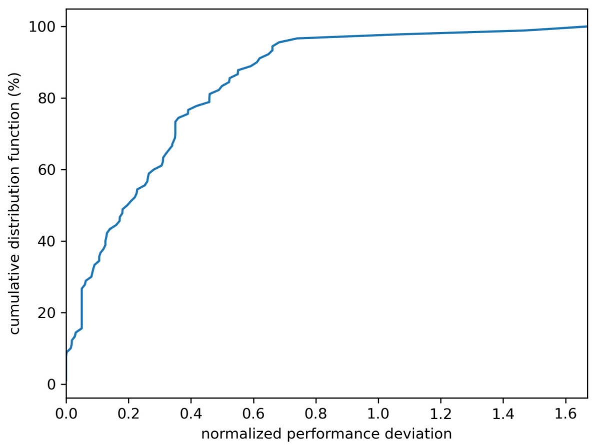


Fig. 12. The CDF of normalized performance deviation was collected from 20 experimental runs. The deviation is 0 when the application performance is not affected by other concurrent running jobs in a shared GPU.

Fig. 13 plots the resource allocation of each inference model over time. Each line in the plot is labeled by its model name followed by the resource requirement ”(request, limit)”. As observed, when there is only one application (Faster R-CNN R50-FPN) running on the GPU initially, it could utilize all the GPU resources. But because the resource limit of the first application is 60%, it can only use 60% resource and achieve 60% performance as we expected. After the second application (RetinaNet R50) joins at time 100, the GPU resource starts to be shared between two applications evenly at 50%. When the third application (Faster R-CNN X101-FPN) joins at time 230, the sum of the minimum resource demand from all three applications reaches 100%, so Gemini has to meet their minimum resource demand as the first priority between time 230 to 330. Finally, after the second application (RetinaNet R50) leaves, the residual capacity is again allocated to the other two applications to maximize the overall GPU utilization. Since the resource limit of the third application (Faster R-CNN X101-FPN) is 40%, its allocation and performance are also bounded at 40%. The residual capacity is allocated to the first application (Faster R-CNN R50-FPN) to ensure the total utilization can still reach 100%. Overall, Gemini can maximize GPU utilization under multi-tenant resource constraints, and resource management can respond to workload changes in a timely manner. Only a small fluctuation ($<5\%$) on resource allocation can be observed over time due to the nature of non-preemptive kernel execution, but we believe it is tolerable to most applications.

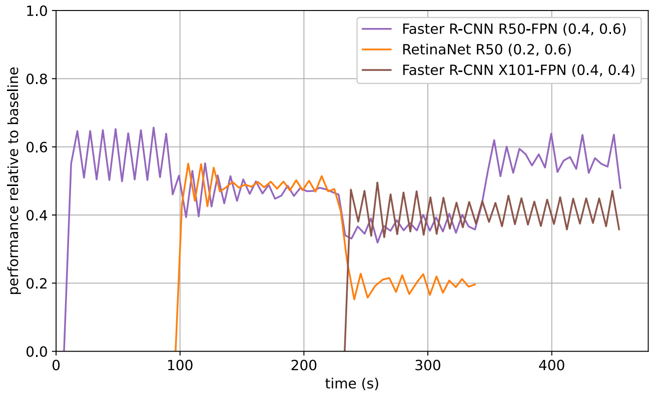


Fig. 13. Elastic resource allocation with applications join and leave dynamically over time.

TABLE 2
The Execution Time and Overuse Amount Comparison Between Dynamic Quota Strategy and Static Quota Strategy for Running a Workload Mixing of Five Applications

Quota (ms)	50	dynamic	100	200
Execution time (s)	3.46	3.188	3.506	3.339
Overuse (ms)	4769.662	1236.1	1854.16	360.69

Quota (ms)	500	750	1000	2000	4000
Execution time (s)	3.328	3.323	3.434	3.8	4.166
Overuse (ms)	874.856	510.004	100.11	70.12	105.938

In sum, Gemini allows users to explore the benefits of GPU sharing to improve system throughput and efficiency and gives resource managers the capability to support multi-tenancy and elastic allocation to ensure user fairness and maximize resource utilization.

4.3 Dynamic Quota Analysis

In this set of experiments, we evaluate the performance improvement of our dynamic quota strategy over the static quota setting and analyze the reasons for the improvements. As discussed, the quota size in our GPU sharing mechanism plays a critical role. When the quota setting is too small, it can cause unfair resource overuse from long-running kernels and higher scheduling overhead. In contrast, when the quota setting is too large, the resource can be wasted when clients hold the token have no ready-to-run kernels. Hence, the setting of the quota should be adjusted according to the runtime behavior of applications.

First of all, we use the workload consisting of 5 different applications for evaluation. We compare the workload execution results from the dynamic quota strategy and the static quota strategy with varied settings from 50 ms to 4000 ms. Table 2 summarizes the workload execution time and the amount of overuse time from each of the settings. We can find that dynamic quota has the shortest execution time in all settings, and its overuse amount is also much lower than the static settings of 50 ms and 100 ms. With a closer look at the dynamic quota strategy, we found the quota size assigned to each application gradually converges to a different value. NAMD often launches long-running kernels, so its quota received at the end is 3300 ms. Detectron2 has the most regular GPU usage pattern controlled by the inference request inter-arrival time, so its quota size equals the request processing time 250 ms. The quota sizes for srad and SOP-GPU are 125 ms and 55 ms, respectively. Finally, the kernel of streamcluster is very small, so it receives the minimum quota size 50 ms. Overall, all applications exhibit some sort of periodic GPU usage pattern, but the length and GPU usage of their kernels are different, the number of kernels merged in a kernel burst is different, and the inter-arrival time between kernel bursts is also different. But our dynamic quota strategy can still roughly capture the behavior of applications and assigns proper quota to each application without human intervention or prior knowledge.

Then, in order to analyze the results of the dynamic quota strategy in more detail, we use the workload of detectron2 alone to compare the dynamic and static strategies

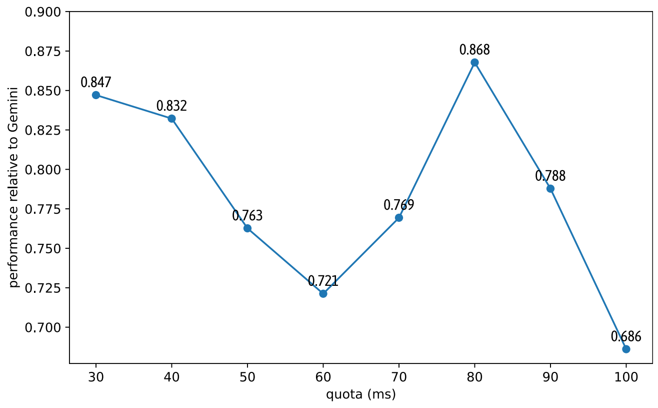


Fig. 14. The performance of the static quota strategy under varied quota size settings for running a detectron2 workload. The result is normalized to the performance of the dynamic quota strategy.

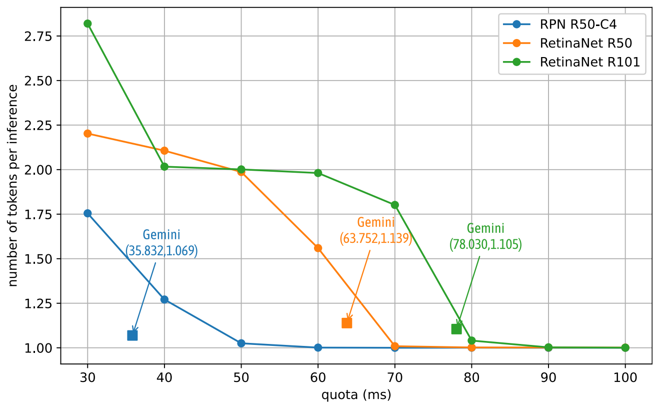


Fig. 16. The average number of tokens needed for processing an inference request. The lines are the results of different static quote size settings. The squares denote the quota and the number of tokens of the dynamic quota strategy for each deep learning model.

because it has the most stable and regular kernel behavior. For the rest of the subsection, we analyze the results collected from running a workload of detectron2 with three different inference models RPN R50-C4, RetinaNet R50, and RetinaNet R101.

The overall performance comparison between dynamic quota scheduling and static quota scheduling is in Fig. 14. The x-axis is the setting of the static quota strategy, and the y-axis is the performance of the static quota strategy normalized to the dynamic quota strategy. Noted, the performance of the dynamic quota strategy is independent of the setting of the static quota. In contrast, the impact of quota size on static quota scheduling is significant, and the impact is hard to be predicted or modeled because of the dynamic workload behavior from mixing jobs. The best performance of the static quota scheduling is observed at quota=80ms, but it is still 23% worse than the performance of the dynamic quota scheduling. The largest improvement from the dynamic quota strategy can reach up to 31%.

To further analyze the results of Fig. 14, we break down the performance improvement of individual application (i.e., deep learning model) in Fig. 15. As observed from the plot, the quota size causes different performance impacts to each application under the static quota strategy. For RetinaNet R101, it has the best performance when the quota is around 80ms. But for RPN R50-C4, has the best performance when the quota is 40 ms. Thus, their performance must be optimized by different quota size settings. But static quota scheduling is agnostic to workload behavior and fixed to all applications. So the aggregated performance from all applications under a given static quota setting cannot result in the best performance. In contrast, Gemini dynamically adjusts quota size based on the kernel burst estimation and workload monitoring information. Therefore, Gemini is able to choose a different quota size for each application. The average quota sizes chosen by Gemini for the three applications are 35.8 ms for RPN R50-C4, 63.8 ms for RetinaNet R50, and 78.0 ms for RetinaNet R101. As shown by the dash lines in the plot, the quota size chosen by Gemini is always closed to the best static quota size observed. Therefore, the result proves that Gemini can achieve higher overall performance improvement without requiring parameter tuning or prior knowledge of applications.

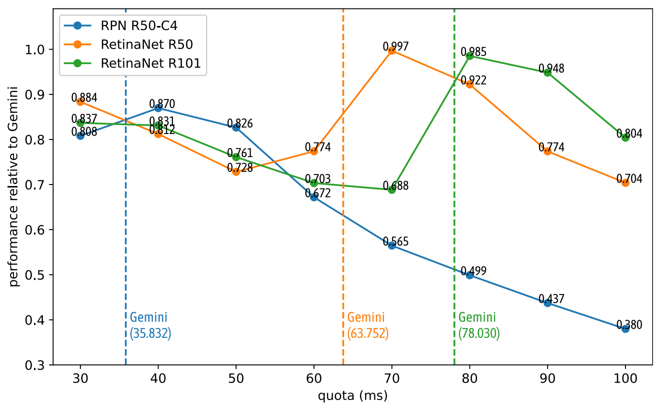


Fig. 15. The performance of the individual application under varied static quota size settings. The result is normalized to the performance of the dynamic quota strategy. The average quote size chosen by the dynamic quota strategy is shown by the dashed line as it is independent of the static quota setting.

Finally, to prove our dynamic quota strategy can capture the execution of each inference request as a kernel burst event. Fig. 16 plots the number of the scheduling events required for processing an inference request under different quota sizes. As expected, the size of the static quota increases as the number of tokens requested for an inference request decreases. But each application reaches one token per request at different quota sizes reflecting the differences in their inference time. In comparison, the average number of tokens per inference request observed for Gemini is very close to 1 for all three applications, which indicates that Gemini does have the ability to adapt runtime workload patterns. The fewer number of scheduling events also helps to reduce coordination overhead between API hooks and the scheduler.

In addition, an accurate kernel burst estimation also means the amount of resource overuse should be reduced. Hence, Fig. 17 compares the amount of overuse under varied static quota size settings for each application. As expected, the amount of resource overuse decreases under a larger static quote size. Eventually, no overuse will occur when the kernel burst time is always shorter than the token quota size. However, a longer token quota is likely to cause the problem of GPU idle and coarse-grained scheduling. Therefore, a shorter token quota with less overuse is preferred. As indicated in Fig. 17, the amount of overuse from Gemini is really close to 0

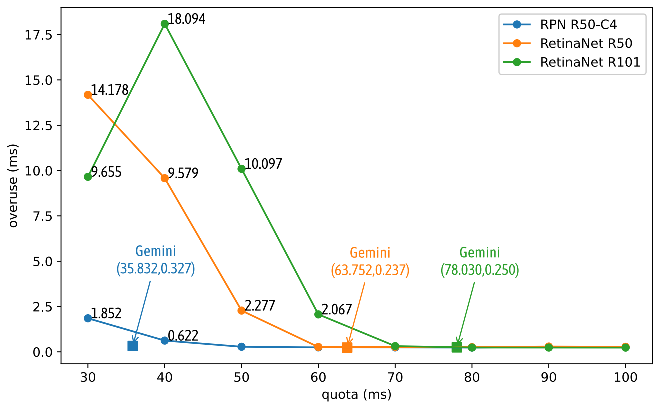


Fig. 17. The average amount of overuse resources from each scheduling timeslice. The lines are the results of different static quote size settings. The squares denote the quota and the overuse amount of the dynamic quota strategy for each deep learning model.

for all the models. Furthermore, as we hoped, the quota size of Gemini is only slightly greater than the minimum quota size required to achieve zero overhead. Interestingly, for the RPN R50-C4 model, we found that Gemini has a lower overuse value than the result of using a smaller static quota size. This indicates we don't always need a larger quota to address the overuse problem. Our dynamic quota selection and token revocation scheme also help to avoid the overuse problem.

4.4 Overhead Evaluation

Finally, an essential requirement of our design is to minimize performance overhead while enabling fine-grained resource management. Hence, we evaluate the performance overhead of our implementation in the following three aspects.

Library Overhead: to measure the performance overhead introduced by API hook and scheduler without GPU sharing, we run a single application on a GPU alone with the Gemini library. The results of detectron2 with different deep learning models are shown in Fig. 18. As observed, the slowdown is less than 3% for all six models, and half of the applications have less than 1% slowdown. Similar slowdown results are observed from other applications as well.

Scalability Overhead: to observe the performance overhead of Gemini under pressure as the number of clients (i.e., applications) shared on a GPU increases. To ensure every client on GPU can receive resources, their request demands are set to $1/n$, where n is the number of clients in the

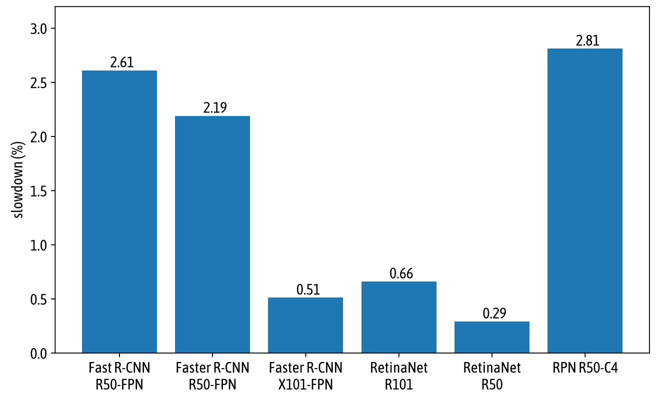


Fig. 18. The performance slowdown of running an application alone on a GPU with the Gemini library.

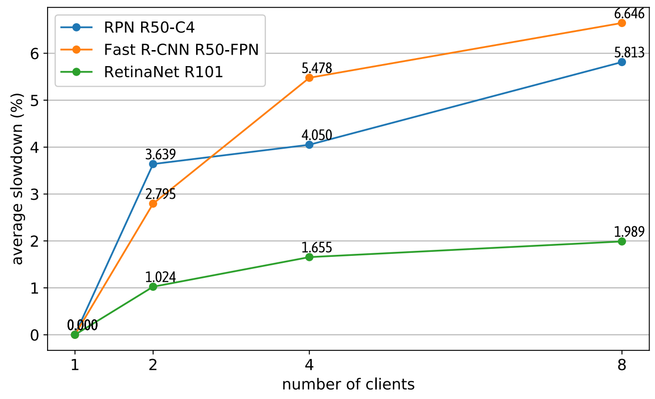


Fig. 19. The performance slowdown of running multiple clients on a shared GPU with Gemini.

experiment. We measure the overall GPU performance as the aggregated performance (i.e., image/sec) from all running applications. The performance measured under one client is considered as the baseline performance here, and we report the performance slowdown under an increasing number of applications varied from 1 to 8. The slowdown results with respect to different models are plotted in Fig. 19. As observed, when GPU is shared, there is about $1 \sim 3\%$ of slowdown initially, then the slowdown slightly increases as more clients are added. But the increasing trend is quite flat, and only 6% slowdown was observed under 8 clients. In practice, most GPU applications demand a high volume of GPU resources with fluctuating workload demand, so we don't expect a single GPU to be shared among too many clients. Therefore, the overhead and performance degradation from Gemini should be acceptable.

Memory Over-Subscription Overhead: when applications allocate a huge amount of memory without launching GPU kernels, applying our GPU sharing technique can improve GPU utilization by consolidating more applications on a single GPU, but memory content may need to be migrated between host and GPU. To evaluate this performance impact, we force the GPU memory usage to increase from 86.25% to 143.70% by adding the number of jobs shared on a GPU. We compare the throughput (number of inference requests per second) of four different optimization strategies. (1) Serial: Serial execution of the jobs without Gemini and GPU sharing. (2) Baseline: Parallel execution of the jobs with Gemini, but no optimization was applied. (3) Resize: empty the unused memory by calling PyTorch API `empty_cache` after each training iteration. (4) Prefetch: prefetch memory content from the host by calling `cudaMemPrefetchAsync` after launch kernel requests received by the backend scheduler. (5) Resize+Prefetch: apply the empty cache and prefetch techniques together. The results normalized to the baseline throughput are plotted in Fig. 20, and we have the following observations.

First of all, the memory over-subscription overhead can diminish the throughput benefits of GPU sharing. When memory is not over-subscribed at 86.25%, the throughput of parallel execution with GPU sharing (baseline) is almost 5 times over the throughput of serial execution. But as the degree of memory over-subscription increases, the overhead of memory transfer for GPU sharing starts to rise, and the improvement from GPU sharing starts to decrease. When the degree of over-subscription reaches 143.70%, the performance of serial

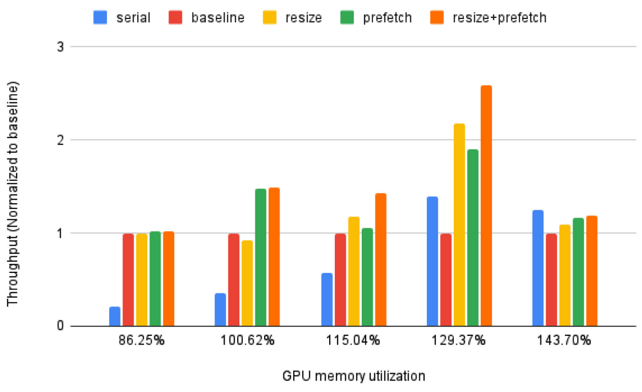


Fig. 20. The performance under memory over-subscription.

execution and parallel execution becomes similar. This result is expected because GPU sharing is used to increase GPU utilization but cannot solve the contention on memory bandwidth. But notice, at memory over-subscription of 129.37%, the serial execution already outperforms parallel execution (baseline) if none of the optimizations is applied. In contrast, with our optimization (resize+prefetch), the throughput of Gemini can still outperform serial execution by 1.86x (=2.59/1.39). Therefore, our optimization can significantly improve the throughput of Gemini under memory over-subscription and extend the range of usability of our approach.

In detail, we found that both prefetch and resize can reduce the data transfer time of unified memory. Especially when the memory is slightly over-subscribed, resize (empty cache) can minimize the amount of data transfer, and the remaining data transfer time could be mostly overlapped by the queuing wait time using prefetching. Therefore, we achieved 43% performance improvement at 115% memory utilization using both techniques together. As the memory is more overly subscribed, more data must be transferred due to higher memory contentions, but size reduction from the empty cache and the overlapping time for prefetching remain the same. As a result, the improvements from both optimizations approaches gradually decrease to 10%. But the performance degradation from unified memory will also remain constant when all the memory contents of a job have to be evicted from the GPU device before each kernel execution.

In sum, although our optimization can minimize the data transfer time comparing to the baseline, we cannot always completely eliminate or hide the data transfer time for memory-intensive applications. So, there is a trade-off between resource utilization and performance, and it is up to the users to decide whether these memory-intensive applications should be run serially on a non-shared GPU or concurrently with degraded performance on a shared GPU. Profiling and monitoring techniques can be used to help users address this issue. But in this work, we only focus on minimizing the data transfer overhead. It will be our future work to develop a performance degradation predictor for users to make the proper decision.

5 Related Work

Container technology has quickly emerged as a replacement for the virtual machine in cloud platforms because of its lightweight and low overhead. However, now is still at the early stage of providing GPU sharing solutions for containers. ConvGPU [12] is a lightweight runtime library installed inside a container using `LD_PRELOAD` to limit its memory allocation on GPU, so GPU can be shared among containers without causing the memory over-subscription problem. GaiaGPU [11] is the closest work to Gemini. It also aims to support multi-tenant and elastic allocation for GPU sharing based on the same scheduling framework and API interception technique. However, it simply allocates a fixed quota for each token request from the applications to distribute the computing resource. The quota is proportional to the resource demand and agnostic to workload patterns, like the static quota scheduling evaluated in our experiments. More important, it didn't discuss or address the resource usage problem from asynchronous kernel execution. So GaiaGPU doesn't track the actual kernel execution time and doesn't require all the resource management techniques proposed in our work, including the kernel burst concept, monitoring technique, or dynamic quota management. As a result, none of the previous approaches can provide the tenancy and performance guarantee like Gemini.

Non-preemptive kernel execution presents a problem for time-sharing GPU because a long-running kernel can monopolize GPU. Therefore, several attempts have been made to support kernel preemption in order to provide fine-grained GPU sharing with the presence of long-running kernels. Most of these work [5], [10], [31] are software frameworks that rely on application code ingestion or modification at compile time to create preemption points in kernel code. Chimera [24] is a hardware solution that introduces streaming multiprocessor (SM) flushing to instantly preempt an SM by detecting and exploiting idempotent execution. However, the solution can only be evaluated on a GPU simulator. In our work, we only consider runtime resource management because compile-time solutions may not be feasible for containerized or cloud environments. Also, our work aims to solve the scheduling overhead problem caused by short-running kernels instead of the fairness problem caused by long-running kernels. Therefore, we don't consider kernel preemption in this work.

There are many studies on the design of schedulers for GPU sharing at OS and driver layers. Rain [29] and its enhanced work, String [28], are both scheduling techniques based on GPU remoting [7], [8], and implement a rich set of scheduling policies to distribute GPU workload across GPU devices in a cluster. In comparison, our work focus on resource sharing problem on a single GPU, and our work is based on API hook, which has much less performance overhead than GPU remoting. PTask [27] is another early work that enables operating systems to directly manage GPU. The PTask API provides a dataflow programming model that asks programmers to express not only where data should move but also how or when. Although the author suggested that the existing CUDA library can be implemented on top of the PTask API, it still requires a significant amount of programming effort. Disengaged [20] implements fair share scheduling on GPU. It uses a runtime profiling technique to estimate the kernel time before scheduling, and it leverages their previously proposed state machine inferencing approach [19] to detect necessary GPU events for scheduling. Hence it needs to insert hooks to Linux kernel and NVIDIA driver's binary interface. TimeGraph [14] is a real-time scheduler that provides isolation and prioritization capabilities for GPU resource management. TimeGraph is implemented at the device driver level based on an open source GPU driver. FELIPE [36] combines time/event-based scheduling policy so that the scheduler can be informed at the end of a kernel execution to perform fine-grained context switch among virtual machines. In sum, we don't expect our approach to out-perform these GPU sharing techniques at OS and driver layers because much less information and control can be obtained at the API layer. But our approach offers transparency and compatibility that cannot be obtained otherwise. As the software stack and hardware model of GPUs evolve at a rapid pace, we believe our solution provides a much general, applicable, and practical solution for practitioners.

Besides sharing GPUs in a time-slicing manner, some studies [23], [26], [31] attempt to share GPU resources in the space domain by running multiple kernels on a GPU at the same time. These techniques control the resource allocation by changing the grid size (number of threads) of kernels so that more GPU cores can be utilized without being over-subscribed. The main challenge of these approaches is that the synchronization and data dependency issues among threads are commonly hard-coded to the grid size. So not all the kernel sizes can be easily modified. The latest work, FLEP [31], proposed a compiler-based kernel transformation technique to allow the kernel size to be reconfigured at runtime.

Finally, more recent works are interested in GPU sharing techniques for specific applications, especially deep learning. By exploring or controlling the GPU usage pattern of applications, the GPU resource can be allocated more efficiently. For instance, Salus [34] and Antman [33] both observed that most of the memory allocation of deep learning jobs are temporal in each training iteration. So if the jobs are context switched on GPUs at the boundary of a training iteration, less aggregated memory space is required. PipeSwitch [3] leverages the layered structure of neural network models and their layer-by-layer computation pattern to pipeline model transmission over the PCIe and task execution in the GPU, and thus the GPU can be shared in a finer-grained by running multiple training and inference jobs at the same time. Last but not least, Zico [16] studied the memory usage variation pattern of deep learning and multiplexes the workload of two deep learning jobs on a single GPU to reduce their aggregated peak memory demand. All these approaches require modifications to the applications or the computing libraries and can only be applied to deep learning applications. In contrast, our approach is general to any GPU application or workload.

6 Conclusion

Supporting GPU sharing is an important and challenging problem. Proprietary GPU stack and asynchronous, non-preemptive kernel make it challenging to provide fine-grained control without introducing additional synchronization overhead. Although this topic has been actively studied in the past several years, existing solutions are mostly based on a custom GPU stack and implemented at the OS or driver layers. In this work, we designed and implemented Gemini, a user-space runtime scheduling framework to enable fine-grained GPU allocation control with low overhead. By introducing the concept of *kernel burst*, we designed a low overhead *event-driven monitor* and a *dynamic time-sharing scheduler*. To the best of our knowledge, Gemini is the only prototype software solution that can be completely transparent to the GPU software stack and agnostic to applications while having the ability to adapt dynamic workload and provide fine-grained resource control. We extensively evaluated our approach using five different GPU workloads. The results prove Gemini can enable multi-tenant and elastic GPU allocation with less limited performance overhead. Comparing to static quota scheduling, Gemini achieved 20%$\sim$30% performance improvement without prior knowledge from applications. The overheads of our approach in terms of library, scalability, and memory over-subscription are also evaluated to prove the applicability and feasibility of our approach.

Overall, the performance benefits of our GPU sharing technique are still affected by the application resource usage behavior, and less improvement can be expected for the applications with the following characteristics. First, our work only explores GPU sharing in the time domain, not the space domain, so that more utilization improvement can be found from applications with small kernel sizes. Second, our work cannot preempt kernel execution, so kernels with unexpected long execution times can still cause the temporal resource overuse problem. The tenancy and fairness are only guaranteed across a longer time period. Third, irregular kernel execution patterns can cause less accurate kernel estimation time and worse performance isolation. Fourth, although our work tends to minimize the data transfer overhead of unified memory, performance degradation and interference problems cannot be eliminated under the memory over-subscription scenario. So, memory-intensive applications can still suffer from the data transfer bottleneck. All these problems could be mitigated by various GPU sharing techniques, such as kernel resizing, kernel preemption, kernel prediction, and memory usage reduction. Unfortunately, the existing solutions mostly require modifications to the OS kernel, GPU driver, or applications. Therefore, in the future, we would like to investigate further the possibility of realizing and integrating these techniques at the GPU API level to extend the usability and benefit of our approach.

References

[1] Linux manual page. Aug. 27, 2021. [Online]. Available: https://man7.org/linux/man-pages/man8/ld.so.8.html

[2] A. M. Aji *et al.*, "On the efficacy of GPU-integrated MPI for scientific applications," in *Proc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2013, pp. 191–202.

[3] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "PipeSwitch: Fast pipelined context switching for deep learning applications," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 499–514.

[4] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization*, 2009, pp. 44–54.

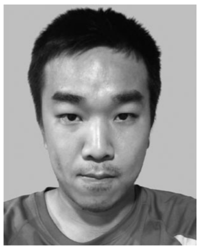
[5] G. Chen, Y. Zhao, X. Shen, and H. Zhou, "EffiSha: A software framework for enabling effficient preemptive scheduling of GPU," in *Proc. 22nd ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2017, pp. 3–16.

[6] H.-H. Chen, E.-T. Lin, and J. Chou, "The GitHub repository of gemini," Aug. 21, 2020. [Online]. Available: https://github.com/NTHU-LSALAB/Gemini

[7] J. Duato, A. J. Peña, F. Silla, R. Mayo, and E. S. Quintana-Ort í, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2010, pp. 224–231.

[8] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A GPGPU transparent virtualization component for high performance computing clouds," in *Proc. Eur. Conf. Parallel Process.*, 2010, pp. 379–391.
[9] Google, "Kubernetes cluster management," 2021. [Online]. Available: http://kubernetes.io/
[10] A. Goswami *et al.*, "GPUShare: Fair-sharing middleware for GPU clouds," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2016, pp. 1769–1776.
[11] J. Gu, S. Song, Y. Li, and H. Luo, "GaiaGPU: Sharing GPUs in container clouds," in *Proc. IEEE Int. Conf. Big Data Cloud Comput.*, 2018, pp. 469–476.
[12] D. Kang, T. J. Jun, D. Kim, J. Kim, and D. Kim, "ConVGPU: GPU management middleware in container based virtualized environment," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 301–309.
[13] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar, "RGEM: A responsive GPGPU execution model for runtime engines," in *Proc. IEEE Real-Time Syst. Symp.*, 2011, pp. 57–66.
[14] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, Art. no. 2.
[15] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class GPU resource management in the operating system," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 401–412.
[16] G. Lim, J. Ahn, W. Xiao, Y. Kwon, and M. Jeon, "Zico: Efficient GPU memory sharing for concurrent DNN training," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 161–175.
[17] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," *Lecture Notes Comput. Sci.*, vol. 8693, pp. 740–755, 2014.
[18] F. Massa and R. Girshick, "maskrcnn-benchmark: Fast, modular reference implementation of instance segmentation and object detection algorithms in PyTorch," 2018. [Online]. Available: https://github.com/facebookresearch/maskrcnn-benchmark
[19] K. Menychtas, K. Shen, and M. L. Scott, "Enabling OS research by inferring interactions in the black-box GPU stack," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2013, pp. 291–296.
[20] K. Menychtas, K. Shen, and M. L. Scott, "Disengaged scheduling for fair, protected access to fast computational accelerators," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2014, pp. 301–316.
[21] NVIDIA, "NVIDIA's next generation CUDA compute architecture: Kepler GK110," *GPU Technol. Conf.*, 2012. [Online]. Available: https://www.nvidia.cn/content/dam/en-zz/Solutions/Data-Center/documents/NV-DS-Tesla-KCompute-Arch-May-2012-LR.pdf
[22] NVIDIA, "Sharing a GPU between MPI processes: Multi-process service (MPS) overview," 2013. [Online]. Available: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf
[23] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU concurrency with elastic kernels," in *Proc. 18th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2013, pp. 407–418.
[24] J. J. K. Park, Y. Park, and S. Mahlke, "Chimera: Collaborative preemption for multitasking on a shared GPU," in *Proc. 20th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2015, pp. 593–606.
[25] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "VGRIS: Virtualized GPU resource isolation and scheduling in cloud gaming," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 2, Jun. 2014, Art. no. 17.
[26] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," in *Proc. 20th Int. Symp. High Perform. Distrib. Comput.*, 2011, pp. 217–228.
[27] C. J. Rossbach, J. Currey, M. Silberstein, B. Ray, and E. Witchel, "PTask: Operating system abstractions to manage GPUs as compute devices," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 233–248.
[28] D. Sengupta, A. Goswami, K. Schwan, and K. Pallavi, "Scheduling multi-tenant cloud workloads on accelerator-based systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 513–524.
[29] D. Sengupta, R. Belapure, and K. Schwan, "Multi-tenancy on GPGPU-based servers," in *Proc. Workshop Virtualization Technologies Distrib. Comput.*, 2013, pp. 3–10.
[30] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why not virtualizing GPUs at the hypervisor?," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 109–120.
[31] B. Wu, X. Liu, X. Zhou, and C. Jiang, "FLEP: Enabling flexible and efficient preemption on GPUs," in *Proc. 22nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2017, pp. 483–496.
[32] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, "Detectron2," 2019. [Online]. Available: https://github.com/facebookresearch/detectron2
[33] W. Xiao *et al.*, "AntMan: Dynamic scaling on GPU clusters for deep learning," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 533–548.
[34] P. Yu and M. Chowdhury, "Salus: Fine-grained GPU sharing primitives for deep learning applications," in *Proc. 3rd MLSys Conf.*, vol. abs/1902.04610, 2020.
[35] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan, "vGASA: Adaptive scheduling algorithm of virtualized GPU resource in cloud gaming," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 11, pp. 3036–3045, Nov. 2014.
[36] X. Zhao, J. Yao, P. Gao, and H. Guan, "Efficient sharing and fine-grained scheduling of virtualized GPU resources," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 742–752.

Hung-Hsin Chen received the BS degree from the Department of Computer Science, National Tsing Hua University (NTHU), Taiwan, in 2020. He conducted research and participated in competitions in the field of high-performance computing and cloud computing under the guidance of Prof. Jerry Chou. He has participated in multiple student competition events held by HPC and HPC-AI advisory councils. He is one of the team members who won the first prize at the Asia Student Supercomputer Challenge, in 2019.



En-Te Lin received the BS degree from the Department of Computer Science, National Tsing Hua University (NTHU), Taiwan, in 2020. He participated in 2019 HPC-AI student competition held by HPC-AI advisory council and won the third prize. After graduation, he is admitted into the MS program at NTHU, and continues his research in high-performance computing under the supervision of Prof. Jerry Chou.

Yu-Min Chou received the bachelor's degree in computer science from National Tsing Hua University (NTHU), Taiwan, in 2020. Currently, he is working toward the PhD degree at National Tsing Hua University (NTHU), Taiwan. His research interests include distributed computing and deep learning. Specifically, he is working on various topics related to federated learning and distributed learning.

Jerry Chou (Member, IEEE) received the PhD degree from the Department of Computer Science and Engineering, University of San Diego (UCSD), San Diego, California, in 2009. He joined the Department of Computer Science, National Tsing Hua University, Taiwan, in 2011 as an assistant professor and was promoted to associate professor, in 2016. Between 2010 and 2011, he was a member of the scientific data management group in Lawrence Berkeley National Lab (LBNL). His research interests are in the broad area of distributed systems including high-performance computing, cloud/edge computing, big data, storage systems, and resource or data management.