MMK: A Hybrid Scheduling Framework for Fine-Grained Multi-Instance GPU Sharing for Deep Learning Applications

***Abstract*—With the rapid growth of deep learning applications and the increasing demand for computational resources, GPU acceleration has become critical for supporting large-scale, compute-intensive deep learning tasks. To improve GPU utilization, emerging GPU sharing technologies such as Multi-Instance GPU (MIG) and Multi-Process Service (MPS) have been widely employed. However, MIG and MPS often suffer from inefficient resource allocation and performance interference. Although integrating MIG with MPS further enhances GPU sharing capability, this approach still incurs complex configuration and coarse-grained scheduling for dynamic workloads, making it ineffective in handling variations in resource demand.**

To address these challenges, we propose MMK, a multi-level GPU sharing system that integrates MIG, MPS, and kernel-level scheduling. First, we design a hybrid scheduler to provide efficient MIG and MPS resource configurations for dynamic workloads. Second, we develop a kernel scheduler to implement fine-grained scheduling strategies that dynamically optimize kernel execution, thereby improving system throughput and reducing resource contention. Extensive experiments demonstrate that compared with the state-of-the-art framework, we improve the average job completion time, makespan, and system throughput by 28%, 32%, and 35%, respectively.

***Index Terms*—GPU Sharing, Deep Learning**

I. Introduction

With the rapid growth of applications such as scientific computing [1], autonomous driving [2], and large language models (LLMs) [3] rapidly expands, GPU has become the core hardware for handling compute-intensive workloads in large data centers [4] due to its highly efficient parallel processing capabilities. However, a single job often underutilizes GPU resources, resulting in significant hardware waste. Consequently, data centers have adopted GPU resource sharing to run multiple jobs concurrently and improve utilization. Nevertheless, collocating multiple jobs introduces severe performance interference, particularly when latency-sensitive online jobs and compute-intensive offline jobs run simultaneously. Therefore, enabling efficient GPU resource sharing that maximizes system throughput while minimizing collocation interference has emerged as a critical challenge for both academia and industry.

Currently, NVIDIA's Multi-Process Service (MPS) [5] enhances GPU utilization by allowing concurrent execution of multiple processes through GPU context multiplexing. However, due to the absence of hardware-level isolation, competition for computing and memory bandwidth among collocation processes still leads to performance interference and violations of Quality-of-Service (QoS). To address this issue, Multi-Instance GPU (MIG) [6] partitions the GPU into multiple fully isolated instances, assigning independent hardware-level resources to each job and eliminating collocation interference. Nevertheless, MIG employs a static partitioning mechanism, which may lead to resource imbalance and incurs non-negligible reconfiguration overhead [7].

Although integrating MIG with MPS provides higher parallelism in resource configuration, it still suffers from configuration complexity for dynamic workloads and coarse-grained scheduling within the MIG partition [8]. Therefore, it is necessary to enable dynamic and efficient GPU resource allocation. However, designing a fine-grained GPU sharing system is non-trivial and faces two challenges.

The first challenge is the complexity and rigidity of MIG and MPS configuration (§II-B). Although their combination provides resource isolation and sharing, limited partition choices and frequent reconfiguration overhead make optimal configuration difficult. As job counts grow, resource contention worsens, leading to imbalance and inefficiency.

The second challenge is the lack of fine-grained scheduling (§II-B). MPS cannot address kernel-level contention within MIG partitions, and preserving performance often requires disruptive reconfigurations. Moreover, MIG and MPS cannot dynamically allocate resources to new online jobs without affecting co-located ones. A kernel-level scheduling strategy is thus needed to ensure QoS under dynamic workloads.

To solve the above challenges, we design a hybrid scheduling framework that integrates MIG, MPS, and kernel-level scheduling to enable efficient GPU partitioning and configuration, while supporting fine-grained resource allocation.

To address the first challenge, we design a hybrid scheduler that leverages MIG's hardware-level isolation and MPS's software-level sharing to provide flexible resource allocation for online and offline jobs. For MPS, we adopt an efficient dynamic configuration strategy that allocates optimal SM resources to dynamically arriving jobs within each partition, enabling rapid adaptation to multi-job resource demands. For MIG, we searches for feasible partition combinations based on the optimal MPS configurations and selects the one that delivers the best overall performance, aiming to reduce resource contention while improving system throughput. To minimize reconfiguration overhead, we prioritize MPS reconfiguration, fully utilize resources within partitions, and reduce the frequency of MIG reconfigurations.

To address the second challenge, we propose a kernel scheduler, which dynamically adjusts the execution order of

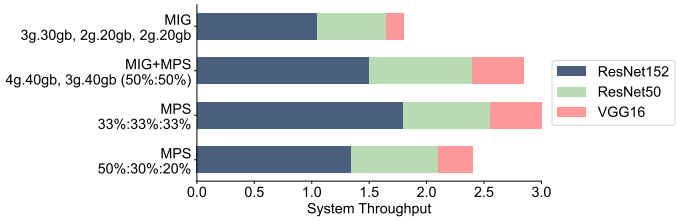

Fig. 1: Performance comparison between different sharing strategies.

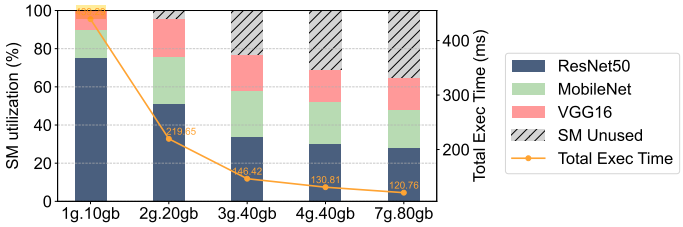

Fig. 2: SM utilization comparison under different MIG partitions.

kernels at the operator level based on real-time resource metrics such as SM occupancy, memory bandwidth utilization, and kernel duration. Specifically, when jobs are launched, we detect kernel-level resource contention. If contention is observed and other online jobs are running, we prioritize kernels from online jobs for SM resource allocation to ensure their QoS. For offline jobs, we adaptively adjust kernel launch timing based on contention intensity to preserve execution efficiency.

To demonstrate the effectiveness of our system, we evaluate our proposed efficient high utilization system. Extensive evaluation proves that the proposed framework can significantly improve ultilization. Our system exceeds 28%, 32%, and 35% in average job completion time, makespan, and system throughput compared with the SOTA GPU scheme.

Our contributions are as follows:

- We have conducted extensive experiments to demonstrate that existing MIG and MPS hybrid methods still suffer from resource contention and low utilization.
- We propose **MMK**, a multi-level GPU sharing system that integrates MIG, MPS, and kernel-level scheduling. **MMK** dynamically allocates resources for co-located jobs, optimizing operator execution to increase throughput and reduce resource contention.
- We have conducted extensive experiments on A100 GPU to demonstrate the effectiveness of **MMK**. **MMK** greatly improves utilization compared with the state-of-the-art framework combining MIG with MPS and improves the average job completion time, makespan, and system throughput by 28%, 32%, and 35%, respectively.

II. Background and Motivation

A. GPU architecture integrated with MIG and MPS

Modern GPU architectures integrate MIG and MPS for efficient execution of deep learning workloads, to further enhance GPU performance. MIG provides hardware-level isolation by partitioning GPU resources such as memory, L2 cache, and DRAM bus, significantly reducing contention. However, MIG partitions are fixed, requiring manual selection, and reconfiguration involves terminating active applications, leading to resource underutilization. MPS operates at the software level, dynamically managing SM allocation, allowing multiple processes to share GPU resources flexibly. Yet, it lacks strict isolation, risking process interference and kernel-level contention. By combining MIG and MPS, modern GPUs achieve a balance between isolation and flexibility, enabling efficient resource utilization across diverse workloads.

B. Observations on Existing Technologies

Observation #1: Designing an effective scheduling strategy that combines MIG and MPS remains challenging.

When combining MIG and MPS for concurrent execution, MIG provides physical isolation while MPS enables job-level parallelism. However, achieving optimal performance remains challenging due to the complexity of resource configuration. To investigate this issue, we evaluate the performance of representative workloads—ResNet152, ResNet50, and VGG16—under different resource allocation strategies. As shown in Figure 1, the four evaluated configurations exhibit significant performance differences. The configuration [4g.40gb, 3g.40gb] outperforms [3g.30gb, 2g.20gb, 2g.20gb], as the former allows ResNet50 and VGG16 to share resources more efficiently, leading to better GPU utilization. Although the latter also provides isolation, its resource utilization is suboptimal. We also observe that system performance is highly sensitive to MPS configurations. The [33%, 33%, 33%] setting achieves higher overall throughput, while [50%, 30%, 20%] significantly degrades performance due to limited SM availability for jobs with low MPS ratios.

Observation #2: Running multiple jobs in MIG partitions with MPS may still result in both underprovision and overprovision of resources.

Although combining MIG with MPS facilitates multi-job parallelism within partitions, it still results in resource overprovisioning or underprovisioning due to fixed partition sizes. To illustrate this issue, we concurrently run three representative workloads, ResNet152, ResNet50, and VGG16, each with 100% MPS on different MIG partitions. Figure 2 shows GPU utilization (left y-axis) and average job completion time (right y-axis) across five configurations. In the 7g.80gb partition, all three jobs are allocated ample resources and achieve the lowest latency, but approximately 30% of GPU resources remain idle. When the partition size is reduced to 4g.40gb, idle SMs decrease and overall GPU utilization improves. As the partition size continues to shrink, resource contention begins

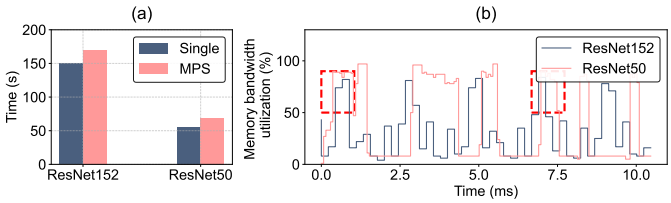

Fig. 3: SM utilization competition in a MIG partition.

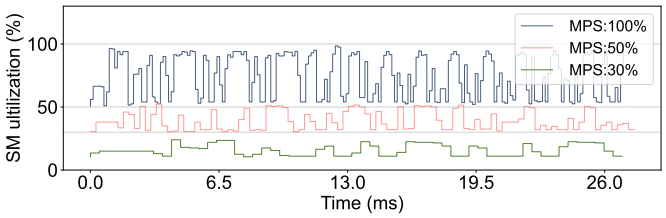

Fig. 4: Job SM utilization changes under different MPS limitations.

to emerge. In the 3g.30gb partition, the average job completion time increases. In the 2g.20gb and 1g.10gb partitions, job performance degrades significantly due to severe contention.

Observation #3: Collocating multiple jobs under MPS still suffers from resource contention

Integrating MIG with MPS to enable parallel execution of multiple jobs within a single MIG partition still results in resource contention, leading to degraded overall resource utilization. In Figure 3(a), we concurrently run inference jobs of ResNet152 and ResNet50 within a 7g.80gb MIG partition, with equal MPS limits set to 50% for both jobs. Compared to executing each job independently on the GPU with the same MPS configuration, the co-execution incurs longer execution time. This indicates that while SM ultilization is constrained via MPS, performance degradation persists due to resource contention.

To further investigate this, we employ Nsight Systems [9] to sequentially profile the jobs with aligned start times (Figure 3(b)). The profiling results reveal that, despite equal SM allocation via MPS, memory bandwidth between kernels also remains a bottleneck due to contention during concurrent execution. Therefore, limiting SM usage alone via MPS is insufficient to eliminate resource contention.

Additionally, analysis with Nsight System shows that under MPS, as the available SM percentage decreases, kernel resource utilization decreases, and the utilization bubble becomes larger. We use ResNet152 as the workload and conduct experiments under different MPS settings. As shown in Figure 4, with the reduction in SM allocation in MPS, kernel resource utilization decreases, and the utilization bubble grows.

III. Design

Figure 5 illustrates the overall architecture of **MMK**, which consists of three core components: a performance predictor, a hybrid partition scheduler, and a kernel scheduler. The performance predictor estimates key performance metrics for each job. The hybrid partition scheduler assigns suitable MIG partitions and configures MPS settings to enable flexible and adaptive resource allocation. The kernel scheduler intercepts CUDA APIs to support fine-grained, operator-level scheduling. During the offline profiling stage, the system analyzes both overall and kernel-level characteristics of offline and online

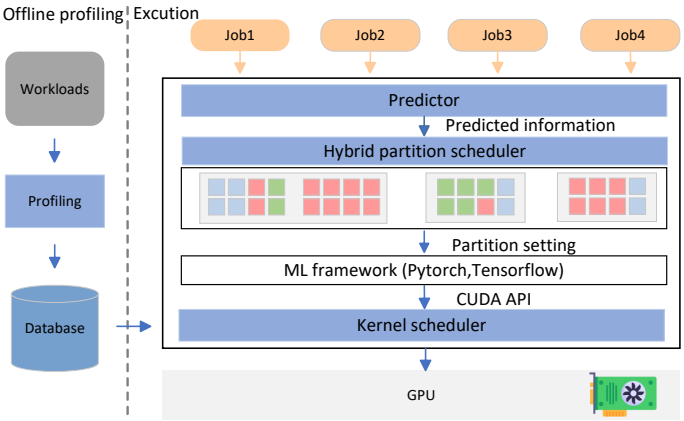

Fig. 5: System overview.

TABLE I: Features used to train predictor

Name	Description
Job type	Job's type, online job or offline job.
MPS configuration	MPS configuration for the job.
Job features	Job's resource features, including SM utilization, memory bandwidth utilization, and L2 cache utilization.
Contention features	Occupied resources in the partition, including SM utilization, memory bandwidth utilization, L2 cache utilization.

jobs to support the performance predictor (§III-A), the hybrid partition scheduler (§III-B), and the kernel scheduler (§III-C). In the execution stage, the performance predictor extracts features from each job based on its resource profile; the hybrid partition scheduler then dynamically adjusts MIG partitions and the number of active SMs per job to maximize throughput; and the kernel scheduler applies kernel-level priority scheduling to guarantee the QoS of online jobs.

A. Performance Predictor

Resource contention is a key factor in job performance degradation. MIG and MPS configurations that ignore contention struggle to ensure system performance. When online jobs are collocated with multiple offline jobs in a resource-constrained environment, contention can cause QoS violations and delays. Offline jobs also experience reduced throughput.

TABLE II: Definitions of variables for Algorithm 1

Name	Description
opt_mig	Optimal MIG configuration
opt_mps	Optimal MPS configuration of all MIG sub-partitionss
max_mig_tp	The maximum throughput of the sum of all MIG sub-partitions
$dict\_job$	The dictionary used to store job
opt_p_tp	Maximum throughput of the sub-partition
opt_p_mps	Optimal MPS configuration of the sub-partition

To address this, we design a contention-aware predictor to estimate job performance within MIG partitions.

We employ the random forest (RF) [10] to predict job performance, as it captures key features and improves generalization through ensemble learning. We use the features in II as inputs, where job features describe resource demands and contention features reflect occupied resources within the partition. The predictor estimates job performance by considering both feature types, using data collected during offline profiling. It outputs throughput for offline jobs and execution time for online jobs. To train the predictor, we generate 1,400 workloads from Table III, using 80% for training and 20% for testing. In §VI-D, we further evaluate the prediction accuracy of the random forest model compared to other commonly used machine learning techniques.

B. Hybrid Partition Scheduler

In this section, we propose the hybrid partition scheduler (HPS), which combines the MIG and MPS to achieve flexible resource partitioning for collocated jobs.

Suppose there is a job list assigned to a GPU, which can be represented as $L = [l_1, l_2, ..., l_k]$. The MPS configuration that constrains a job's SM utilization is defined as $ML = [10, 20, 30, 40, .., 100]$. A MIG partition combination can be represented as $pc = [p_1, p_2, ..., p_n]$, where n represents the number of subpartitions in the MIG combination. The partition $p_i \in \{1g.10gb, 2g.20gb, 3g.40gb, 4g.40gb, 7g.80gb\}$ represents five different types of partitions determined by NVIDIA. All possible configuration partition list can be represented as $PL = [pc_1, pc_2, ..., pc_m]$, where m represents the total number of possible MIG partitions. In Algorithm 1, we sort jobs by memory size and initialize a partition combination pc (line 1). If the current partition has sufficient memory, jobs are greedily assigned to it (line 5). To ensure resource guarantees for both online and offline jobs while minimizing MIG reconfiguration overhead, we prioritize MPS reconfiguration. When jobs arrive or complete, we update the MPS settings for offline jobs (line 6).

Specifically, we first sort the jobs in descending order based on the ratio of throughput to SM utilization, prioritizing jobs with more efficient SM ultilization (line 25). Then, we use the function `getRemainSM` to retrieve the remaining available SMs in the current partition (line 26). We allocate SMs to jobs sequentially according to the sorted order and use variables `opt_p_tp` and `opt_p_mps` to record the current partition's best throughput and corresponding MPS configuration (lines 27-34). The function `stp` estimates the throughput of each job under a given configuration using the performance predictor. To provide the optimal MIG partition, we perform MIG reconfiguration only when offline jobs are running (line 12). We iterate over all candidate partition combination lists (line 13). For each partition combination $PL[k]$, we compute the sum of throughputs using the function `MPSconfig` and collect the optimal MPS configuration for each sub-partition (line 14). We select the partition combination with the highest throughput max_mig_stp and optimal MPS configuration opt_mps as the optimal configuration opt_mig, to maximize offline job

Algorithm 1: MIG combined MPS configuration algorithm

```
1  Sort L by memory
2  Initialize a partition combination pc
3  while i < len(L) do
4      for j < len(pc) do
5          if PC[j] is meet L[i] then
6              _,opt_mps ←MPSConfig(P_Job)
7              Append L[i] to dict_job[pc[j]];
8              dict_job[pc[j]] ← L[i]
9              Reconfigure opt_mps for the jobs_off
10         end
11     end
12     if online job is none then
13         for k to len(PL) do
14             pc_tp,pc_mps ← Σ_{m=0}^{len(PL[k])}
                MPSConfig(dict_job[PL[k][m]], PL[k])
15             if stp > max_mig_stp then
16                 max_mig_stp ← pc_stp
17                 opt_mig ← PL[k]
18                 opt_mps ← pc_mps
19             end
20         end
21         Reconfigure MIG and MPS to the opt_mig and
            opt_mps
22     end
23 end
24 Function MPSConfig(P_Job):
25     Sort jobs by throughput/SM utilization in
        descending order
26     rsm ←getRemainSM()
27     for i = 1 to len(jobs) do
28         if rsm < 0 then
29             return opt_p_tp, opt_p_mps
30         end
31         Allocate jobs[i].sm to jobs[i]
32         rsm ← rsm − jobs[i].sm
33         opt_p_tp ← opt_p_tp + stp(jobs[i])
34         opt_p_mps[jobs[i]] ← Compute MPS
            configuration of jobs[i]
35     end
```

```
1  def run_scheduler(jobs_online, jobs_offline):
2      schedule = True
3      while True:
4          job_online, job_offline = jobs_online.peek()
               , jobs_offline.peek()
5          ava_res = get_available_resources()
6          if job_online != None:
7              for kernel_online in job_online:
8                  if check_contention(kernel,ava_res):
9                      launch_kernel(kernel_online)
10                     dur = kernel_online.time
11                 else :
12                     schedule = False
13             jobs_online.pop()
14             avail_res=updateRes(avail_res)
15         if job_offline != None:
16             Thres=DURATION / job_offline.MPS_limit
17             for kernel_offline in job_offline:
18                 if schedule:
19                     and !check_contention(
                           kernel_offline,ava_res):
20                     launch_kernel(kernel_offline)
21                 else if wait_time > Thres:
22                     launch_kernel(kernel_offline)
23                     wait_time = 0
24                 else :
25                     wait_time += dur
26             jobs_offline.pop()
27             avail_res=updateRes(avail_res)
28
29 def check_contention(kernel, avail_res):
30     if kernel.req_sm <= avail_res.sm and
31        kernel.req_shared_mem <= avail_res.mem and
32        kernel.req_bandwidth <= avail_res.bandwidth:
33        return True
34     else: return False
```

Listing 1: Kernel scheduling algorithm

performance (lines 15–19). Finally, we reconfigure MIG and MPS (line 21).

C. Kernel Scheduler

In this section, we design a kernel scheduler to achieve fine-grained resource scheduling.

Pseudocode shown in Listing 1 illustrates **MMK**'s scheduling strategy. During scheduling, **MMK** first retrieves the current availability of GPU resources, including the number of SMs, memory capacity, and bandwidth (line 5). Based on this information, the system determines whether multiple kernels can be executed in parallel without exceeding resource constraints. Each job's kernel is characterized by its resource requirements and execution duration, which are leveraged for kernel-level scheduling and resource contention analysis.

For each job in the online job queue, **MMK** sequentially examines all kernels of the current job, verifying whether their resource requirements can be met. If sufficient resources are available, the kernel is launched. To enable accurate tracking of online kernel durations, the system records the execution time of each online kernel (lines 7–10). If the available resources are insufficient to meet a kernel's requirements, a scheduling flag is set to `False`, thereby preventing offline jobs from contending for resources (line 12).

Furthermore, if offline jobs are present in the queue, **MMK** first calculates a waiting-time threshold, denoted as `Thres`, for each offline job (line 16). `Thres` is computed based on a constant `DURATION` and the job's `MPS_limit`. The value of `MPS_limit` is determined by the hybrid partition scheduler and represents the optimal MPS configuration for the current offline job. A smaller `MPS_limit` results in a larger `Thres`, reflecting higher contention within the current partition and thus longer waiting times for offline jobs. Conversely, a larger `MPS_limit` leads to a smaller `Thres`, indicating more available resources and increased opportunities to launch offline kernels. Subsequently, we check the remaining available resources to determine whether resource contention exists for offline jobs. If sufficient resources are available, the offline kernel is launched (lines 18–20). Additionally, if an offline job is in a waiting state and its waiting time exceeds the threshold `Thres`, the kernel is also launched (lines 22–23). If the execution requirements of an offline kernel cannot be met, the system accumulates the execution time of online kernels to update the value of `Thres` (line 25).

IV. Implementation

We implement **MMK** in approximately 3,000 lines of C++ and Python code. The system is integrated as a dynamically linked library, enabling support for deep learning frameworks such as PyTorch and TensorFlow by intercepting CUDA API calls. For hybrid partition scheduling, **MMK** leverages both MIG and MPS APIs. MIG partitions are configured by selecting from predefined partition sizes using NVIDIA-provided commands, and we pre-record the UUIDs of all possible partition configurations to eliminate runtime overhead. For MPS, we use the environment variable `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` to control SM usage per job, and employ `CUDA_MPS_PIPE_DIRECTORY` to specify communication paths, enabling the launch of independent MPS control daemons for each GPU partition. To enable kernel-level scheduling, we intercept CUDA API calls at the operator level, collecting runtime metrics such as SM utilization, memory bandwidth, and memory consumption. These metrics are used to predict job priorities and control the timing of API releases accordingly.

V. Methodology Evaluation

Evaluation Setup: The experimental setup includes two Intel Xeon Platinum 8369B CPU with 64 threads and NVIDIA A100-PCIe 80GB GPUs.

Workloads: We follow the workloads settings of MISO [7] and generate a job trace based on Helios [4]. For the testbed evaluations, we create 40 online and 20 offline jobs from Table III. The job arrival intervals follow a Poisson distribution with λ = 40 seconds. For the simulation experiments, we configure 800 online and 400 offline jobs.

TABLE III: Workloads used to evaluate MMK

Name	Workload	Batch size
VGG16 [11]	Inference	32, 64, 128, 256
MobileNet [12]	Inference	32, 64, 128, 256
ResNet18 [13]	Inference	32, 64, 128, 256
BERT [14]	Train	2, 4, 8, 16
ResNet152 [13]	Train	16, 32, 64, 128
DenseNet201 [15]	Train	16, 32, 64, 128
Transformer [16]	Inference and train	16, 32, 64, 128
DenseNet121 [15]	Inference and train	32, 64, 128, 256
VGG19 [11]	Inference and train	32, 64, 128, 256
ResNet50 [13]	Inference and train	32, 64, 128, 256

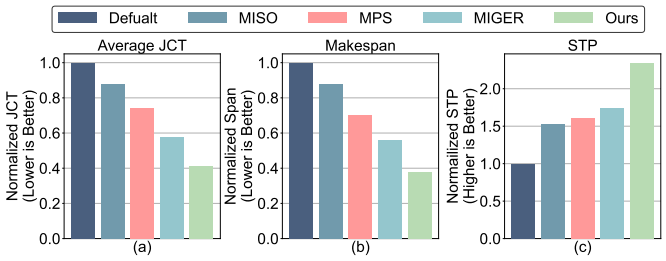

Fig. 6: Performance comparison between competing strategies. All results are normalized to Default.

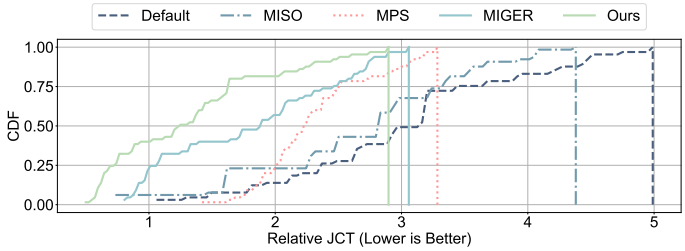

Fig. 7: The CDF of relative JCT for individual jobs is shown, where each job's JCT is normalized to the JCT when running on an exclusive A100 GPU without queuing delay. The vertical line indicates the maximum degradation value.

Baselines and metrics: We compare **MMK** with several baselines. Default runs all jobs without optimization, while MPS executes them in parallel without SM constraints. MISO [7] minimizes completion time when jobs fit available partitions; otherwise, it applies FCFS and queues excess jobs. MIGER [8] is adapted to multi-job scenarios by packing jobs into MIG partitions and tuning MPS settings for performance. We evaluate all methods using three metrics: average job completion time (JCT), system throughput (STP), and makespan, which measure job latency, processing capacity, and worst-case delay, respectively.

VI. Evaluation

A. Performance Comparison

Figure 6 presents the performance comparison across five scheduling strategies. MPS reduces JCT by 25% over Default due to its parallel execution capability, while MISO incurs 12% higher JCT than MPS because of increased queuing delays. MIGER improves upon MISO by isolating high-SM-demand jobs, enabling partial concurrency. **MMK** achieves the best overall performance, reducing JCT by 47% compared to Default and 28% over MIGER through kernel-level priority scheduling. **MMK** also outperforms all baselines in makespan and STP. It reduces makespan by 32% and improves STP by 35% over MIGER by enabling finer-grained resource allocation and operator-level scheduling, which mitigates contention and alleviates long-tail job impact.

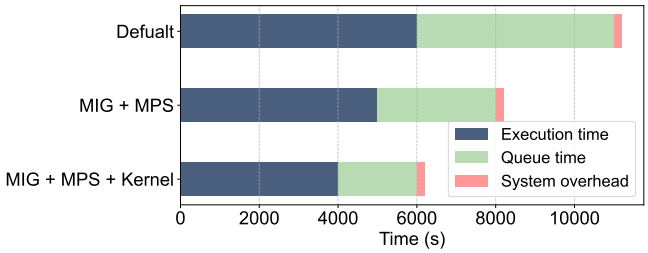

Fig. 8: Performance breakdown analysis of MMK.

Figure 7 further compares the JCT performance of all methods, normalized to Default and presented as a cumulative distribution function (CDF). The results highlight **MMK** 's effectiveness in multi-job scenarios. While MIGER significantly reduces JCT and improves throughput over MPS and MISO, **MMK** further improves GPU utilization and reduces fragmentation by dynamically adjusting MIG partitions and MPS configurations based on workload characteristics, combined with kernel-level scheduling.

B. Performance Analysis Breakdown

We conduct a performance breakdown analysis of the key components of **MMK**. Specifically, we adopt a fixed 7g.80gb MIG partition as the baseline configuration without reconfiguration, and progressively introduce MIG + MPS optimizations and kernel-level enhancements. We evaluate their impact on execution time, queuing delay, and system overhead. As shown in Figure 8, the baseline configuration exhibits substantial queuing delays due to resource contention. Incorporating MIG + MPS alleviates contention through dynamic partitioning and enables concurrent execution within each partition via MPS, thereby reducing both queuing and execution times. The configuration with MIG, MPS, and kernel scheduling further applies fine-grained priority scheduling and resource allocation, mitigating conflicts in parallel execution and achieving additional reductions in latency and overhead.

Figure 9(a) and Figure 9(b) compare the performance of **MMK** under identical MPS configurations. Specifically, the MPS setting for the ResNet152 training job is set to 50% and 100%, respectively, while the ResNet50 inference job consistently uses the 100% MPS setting. Compared to the configuration without kernel scheduling, **MMK** achieves a 2× improvement in the execution time of the ResNet50 inference

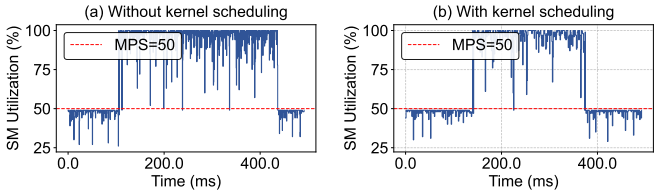

Fig. 9: SM utilization comparison between with vs. without kernel scheduling.

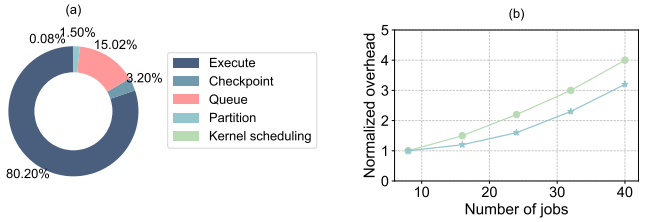

Fig. 10: Overhead breakdown analysis. (a) shows the overhead breakdown during the scheduling process of MMK. (b) shows the overhead changes of partition and KS. numbers of jobs.

job. This improvement is attributed to **MMK** 's priority-based scheduling, which prioritizes resource allocation for online jobs to ensure QoS.

C. System Overhead

In this subsection, we analyze the overhead introduced by two key components of **MMK**: dynamic MIG reconfiguration and kernel-level scheduling.

MIG and MPS reconfiguration overhead: To adapt diverse workload demands and improve GPU utilization, **MMK** reconfigures MIG partitions and MPS settings at runtime. These operations introduce overhead from partition reconfiguration and checkpointing when restarting offline jobs via the MPS server. As shown in Figure 10(a), MIG reconfiguration accounts for only 1.5% of the workload lifecycle, thanks to the hybrid scheduler's ability to minimize reconfiguration frequency by densely packing jobs. In contrast, checkpointing incurs a slightly higher overhead of 3.2%, due to dynamic MPS adjustments that rebalance resources among offline jobs as workloads arrive or complete.

Kernel scheduling overhead: To improve GPU utilization and prioritize online jobs at the kernel level, **MMK** dynamically adjusts the execution priority of collocated kernels based on job-specific requirements. As shown in Figure 10(a), the average scheduling overhead is only 0.08% of total execution time, achieved by prioritizing online jobs and limiting the execution durations of offline jobs. This microsecond-level overhead demonstrates the efficiency of the mechanism with minimal performance impact. Figure 10(b) further shows that the kernel scheduler adapts priorities under dynamic workloads without introducing noticeable overhead, confirming that **MMK** 's fine-grained scheduling remains both effective and lightweight.

TABLE IV: Prediction Accuracy of Different ML Techniques.

Model	Absolute Error	Relative Error
SVR	27.2%	27.08%
LR	21.7%	20.01%
MLP	11.1%	4.65%
RF	3.60%	3.51%

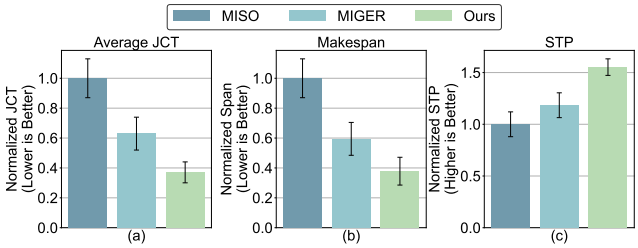

Fig. 11: Performance comparison in simulation.

D. Performance Prediction Accuracy

We chose three representative algorithms in machine learning for comparison: linear regression (LR) [17], multilayer perception (MLP) [18], and support vector regression (SVR) [19]. At the same time, we choose the absolute error that can most intuitively express the prediction performance and the relative error that can better reflect the deviation from the actual value as the accuracy evaluation metric. The result is shown in Table IV. RF achieves the best performance in terms of absolute error, 3×, 6×, and 7× lower than MLP, LR, and SVR, respectively. It is also superior to the other three methods in terms of relative error.

E. Simulation Evaluation

We evaluate **MMK** in a large-scale multi-GPU simulation. As shown in Figure 11, **MMK** achieves 41.18%, 36.36%, and 31.11% improvements over MIGER in JCT, makespan, and system throughput, respectively, benefiting from fine-grained offline job isolation and kernel-level priority scheduling. While MIGER improves over MISO by enabling intra-partition concurrency and reducing SM contention, MISO suffers from higher queuing delays due to its lack of intra-partition parallelism. Under high concurrency, the Default baseline performs the worst due to severe resource contention. Overall, **MMK** remains robust under large-scale workloads by improving parallelism, reducing contention, and significantly enhancing job latency and throughput.

VII. Related Work

We summarize current approaches for temporal and spatial GPU sharing, highlighting their limitations in fine-grained resource allocation and interference management, and demonstrate how our method improves GPU utilization. **Temporal sharing.** Gandiva [20] adds intra-job predictability and dynamic job migration for better efficiency. AntMan [21] coordinates memory and computation for efficient multi-job execution in clusters. KubeShare [22] treats GPUs as first-class

resources in Kubernetes, improving allocation efficiency. Gemini [23] supports multi-tenancy and elastic GPU allocation. TGS [24] ensures high utilization and performance isolation with adaptive rate control. However, existing methods fail to provide optimized scheduling for parallel tasks. We propose a priority-based scheduling approach at the kernel level for parallel execution of multiple jobs.

Spatial sharing. MIGER [8] combines MPS and MIG for dynamic partition sizes and efficient resource allocation. MIG-serving [25] uses genetic algorithms for efficient partitioning in DNN inference. Paris [26] combines partitioning and elastic scheduling for multi-GPU inference. MIG struggles with agility due to long partition creation times. iGniter [27] employs MPS to provision GPU resources for cloud-based DNN inference jobs. FaST-GShare [28] extends MPS for spatio-temporal sharing in serverless inference, improving throughput. Fine-grained kernel scheduling, seen in REEF [29] and Orion [30], reduces interference but still faces trade-offs between isolation and efficiency. Previous methods do not offer fine-grained resource allocation and scheduling. We present a flexible resource configuration and kernel-level scheduling solution.

VIII. Conclusion

In this paper, we present **MMK**, an efficient high utilization system for GPU sharing. It integrates MIG, MPS, and kernel-level scheduling. By achieving flexible partition settings and implementing fine-grained kernel scheduling strategies, it dynamically optimizes operator execution, increasing system throughput and reducing resource contention. The proposed framework proves its effectiveness in extensive experiments. Compared with MIGER scheme, we improve the average JCT, makespan, and STP by 28%, 32%, and 35%, respectively.

References

[1] J. Ott, M. Pritchard, N. Best, E. Linstead, M. Curcic, and P. Baldi, "A fortran-keras deep learning bridge for scientific computing," *Scientific Programming*, vol. 2020, no. 1, p. 8888811, 2020.

[2] L. Chen, Y. Zhang, B. Tian, Y. Ai, D. Cao, and F.-Y. Wang, "Parallel driving os: A ubiquitous operating system for autonomous driving in cpss," *IEEE Transactions on Intelligent Vehicles*, vol. 7, no. 4, pp. 886–895, 2022.

[3] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[4] Q. Hu, P. Sun, S. Yan, Y. Wen, and T. Zhang, "Characterization and prediction of deep learning workloads in large-scale gpu datacenters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

[5] "Nvidia multi-process service (mps)," 2023.

[6] "Nvidia multi-instance gpu (mig) user guide." https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html, 2023.

[7] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "Miso: exploiting multi-instance gpu capability on multi-tenant gpu clusters," in *Proceedings of the 13th Symposium on Cloud Computing*, pp. 173–189, 2022.

[8] B. Zhang, S. Li, and Z. Li, "Miger: Integrating multi-instance gpu and multi-process service for deep learning clusters," in *Proceedings of the 53rd International Conference on Parallel Processing*, pp. 504–513, 2024.

[9] NVIDIA Corporation, "NVIDIA Nsight Systems." https://developer.nvidia.com/nsight, 2024. Accessed: 2025-02-07.

[10] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[12] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[14] J. D. M.-W. C. Kenton and L. K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of naacL-HLT*, vol. 1, Minneapolis, Minnesota, 2019.

[15] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2016.

[16] A. Vaswani, "Attention is all you need," *Advances in Neural Information Processing Systems*, 2017.

[17] S. Weisberg, *Applied linear regression*, vol. 528. John Wiley & Sons, 2005.

[18] A. Pinkus, "Approximation theory of the mlp model in neural networks," *Acta numerica*, vol. 8, pp. 143–195, 1999.

[19] M. Awad, R. Khanna, M. Awad, and R. Khanna, "Support vector regression," *Efficient learning machines: Theories, concepts, and applications for engineers and system designers*, pp. 67–80, 2015.

[20] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang, *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proceesings of the USENIX Symposium on Operating Systems Design and Implementations (OSDI)*, 2018.

[21] W. Xiao, S. Ren, Y. Li, Y. Zhang, P. Hou, Z. Li, Y. Feng, W. Lin, and Y. Jia, "Antman: Dynamic scaling on gpu clusters for deep learning," in *Proceesings of the USENIX Symposium on Operating Systems Design and Implementations (OSDI)*, 2020.

[22] T.-A. Yeh, H.-H. Chen, and J. Chou, "Kubeshare: A framework to manage gpus as first-class and shared resources in container cloud," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, 2020.

[23] H.-H. Chen, E.-T. Lin, Y.-M. Chou, and J. Chou, "Gemini: Enabling multi-tenant gpu sharing based on kernel burst estimation," *IEEE Transactions on Cloud Computing*, vol. 11, no. 1, pp. 854–867, 2021.

[24] B. Wu, Z. Zhang, Z. Bai, X. Liu, and X. Jin, "Transparent {GPU} sharing in container clouds for deep learning workloads," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 69–85, 2023.

[25] C. Tan, Z. Li, J. Zhang, Y. Cao, S. Qi, Z. Liu, Y. Zhu, and C. Guo, "Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem," *arXiv preprint arXiv:2109.11067*, 2021.

[26] Y. Kim, Y. Choi, and M. Rhu, "Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 607–612, 2022.

[27] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang, Z. Zhou, and F. Liu, "igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 3, pp. 812–827, 2022.

[28] J. Gu, Y. Zhu, P. Wang, M. Chadha, and M. Gerndt, "Fast-gshare: Enabling efficient spatio-temporal gpu sharing in serverless computing for deep learning inference," in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 635–644, 2023.

[29] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 539–558, 2022.

[30] F. Strati, X. Ma, and A. Klimovic, "Orion: Interference-aware, fine-grained gpu sharing for ml applications," in *Proceedings of the Nineteenth European Conference on Computer Systems*, pp. 1075–1092, 2024.