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

ZHUOLONG JIANG, Shanghai Jiao Tong University, Shanghai, China
ZINUO CAI, Shanghai Jiao Tong University, Shanghai, China
YAWEN LI, Beijing Simulation Center, Beijing, China
ZIZONG WANG, China Petrochemical Corporation, Beijing, China
RUHUI MA, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
HAIBING GUAN, School of Computer Science, Shanghai Jiao Tong University, Shanghai, China
BUYYA RAJKUMAR, The University of Melbourne, Melbourne, Australia

With the rapid growth of deep learning applications and their 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 suffers from inefficient resource allocation while MPS introduces performance interference. Although integrating MIG with MPS further enhances GPU sharing capability, this approach still requires complex configuration and coarse-grained scheduling for dynamic workloads, making it ineffective in handling variations in resource demand for online jobs and offline jobs. To address these challenges, we propose **MMK**, a multi-level GPU sharing system that integrates MIG, MPS, and kernel-level scheduling for online and offline jobs. First, we design a hybrid scheduler to efficiently configure MIG and MPS resources for dynamic job demands. 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 reduce the average job completion time and makespan by 28% and 32%, respectively, and increase system throughput by 35%.

CCS Concepts: • **Software and its engineering** → **Scheduling**; • **Computer systems organization** → *Parallel architectures;*

Additional Key Words and Phrases: GPU sharing, deep learning, scheduling, resource management

ACM Reference Format:
Zhuolong Jiang, Zinuo Cai, Yawen Li, Zizong Wang, Ruhui Ma, Haibing Guan, and Buyya Rajkumar. 2026. MMK: A Hybrid Scheduling Framework for Fine-Grained GPU Sharing for Deep Learning Applications. *ACM Trans. Arch. Code Optim.* 23, 3, Article 84 (July 2026), 25 pages. https://doi.org/10.1145/3820163

Authors' Contact Information: Zhuolong Jiang, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: jzz19961996@sjtu.edu.cn; Zinuo Cai, Shanghai Jiao Tong University, Shanghai, China; e-mail: kingczn1314@sjtu.edu.cn; Yawen Li, Beijing Simulation Center, Beijing, Beijing, China; e-mail: liyawen0026@163.com; Zizong Wang, China Petrochemical Corporation, Beijing, Beijing, China; e-mail: wangzz@sinopec.com; Ruhui Ma (corresponding author), School of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: ruhuima@sjtu.edu.cn; Haibing Guan, School of Computer Science, Shanghai Jiao Tong University, Shanghai, Shanghai, China; e-mail: hbguan@sjtu.edu.cn; Buyya Rajkumar, The University of Melbourne, Melbourne, Victoria, Australia; e-mail: rbuyya@unimelb.edu.au.

ACM 1544-3973/2026/07-ART84
https://doi.org/10.1145/3820163

1 Introduction

With the rapid growth of applications such as autonomous driving [5, 32, 36], cloud computing [18, 23], edge computing [9, 35] and **large language models** (**LLMs**) [1, 3, 9, 16, 24], GPUs have become a computing resource supporting compute-intensive workloads in large datacenters [12–14, 17] due to their highly efficient parallel processing capabilities. However, a single job often underutilizes GPU resources, resulting in significant hardware waste. Consequently, data centers have adopted GPU 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.

Currently, NVIDIA's **Multi-Process Service** (**MPS**)[1] 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**)[2] 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 can incur non-negligible reconfiguration overhead [22].

Combining MIG and MPS can address their respective shortcomings by leveraging the isolation features of MIG and the multiplexing capabilities of MPS. Prior work MIGER [34] combines MIG and MPS to execute two jobs in parallel within a partition, aiming to improve system performance. However, MIGER still suffers from configuration complexity under dynamic multi-jobs and coarse-grained scheduling within the MIG partition. 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.

- **Challenge 1: Complexity and inflexibility of MIG and MPS configuration** (see Observations 1–3 in Section 2.2). Although the combination of MIG and MPS provides resource isolation and sharing, limited partition choices and frequent reconfiguration overhead make optimal configuration difficult. Increasing job counts intensify resource contention, which degrades system efficiency.
- **Challenge 2: Lack of fine-grained scheduling** (see Observation 4 in Section 2.2). MPS primarily controls the percentage of **Streaming Multiprocessors** (**SM**) allocated to each job while overlooking other performance-critical factors (e.g., memory bandwidth). In addition, MPS settings are immutable after a process starts, so kernel-level contention persists among collocated online and offline jobs. Alleviating kernel-level resource competition by reconfiguring MPS typically disrupts running jobs and incurs high restart/reconfiguration overheads.

To address the above challenges, we design **MMK**, a system that integrates the hardware-level isolation of MIG, the elastic resource sharing capability of MPS, and fine-grained kernel-level scheduling to enable multi-level GPU resource management. **MMK** can adaptively generate optimal partition configurations for dynamic multi-job workloads while reducing reconfiguration overhead to improve overall system performance. In addition, **MMK** dynamically optimizes the execution timing of kernels. When resource contention occurs, it prioritizes online jobs while maintaining the performance of offline jobs to ensure QoS.

[1] https://docs.nvidia.com/deploy/mps/index.html
[2] https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

In specific, we design a hybrid scheduler that leverages MIG's hardware-level isolation and MPS's software-level sharing to address the first challenge. Instead of relying on static rules or exhaustive searches, our approach aligns MIG partition granularity with the resource heterogeneity of workloads to identify efficient spatial configurations. Complementing this, we enhance MPS with an elastic oversubscription mechanism that dynamically exploits transient resource slack within partitions. By coordinating these two levels of resource management and optimizing reconfiguration timing, **MMK** reduces resource fragmentation and contention overhead while maximizing overall throughput.

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

In summary, our contributions are as follows:

- We identify two key challenges in scheduling dynamic multiple workloads with MIG or MPS through extensive experiments.
- We propose **MMK**, a multi-level GPU sharing system that integrates MIG, MPS, and kernel-level scheduling. It provides an efficient hybrid configuration scheme combining MIG and MPS for dynamic multi-job workloads and supports fine-grained kernel-level resource scheduling to improve GPU utilization.
- We conduct extensive experiments on two A100 GPUs to demonstrate the effectiveness of **MMK**. Compared with the state-of-the-art framework, **MMK** achieves 28% and 32% lower average job completion time and makespan, and 35% higher system throughput than the state-of-the-art framework.

2 Background and Motivation

2.1 GPU Sharing

Multi-Process Service MPS is a software-level technique that allows multiple processes to concurrently share a single GPU by managing SM allocation. It dynamically adjusts compute resources based on user-defined quotas, enabling efficient execution of collocated jobs while mitigating resource conflicts. In practice, MPS facilitates resource control via environment variables. However, as it does not enforce full physical isolation, processes may still interfere with each other, especially under unbalanced resource demands. Although MPS provides flexible compute resource allocation, it cannot eliminate kernel-level contention.

Multi-Instance GPU. MIG is an NVIDIA GPU virtualization technology that provides hardware-level resource isolation by partitioning components such as the memory subsystem, crossbar, L2 cache, memory controller, and DRAM bus. This isolation significantly reduces resource contention in collocation scenarios. However, MIG supports only a limited set of partition sizes (as shown in Table 1, the A100 80GB offers five fixed configurations), necessitating manual selection of partition combinations to accommodate varying job requirements. Reconfiguring MIG requires terminating all active applications, leading to resource underutilization and degraded system throughput. Moreover, the inflexible MIG partitions may result in suboptimal resource allocation, causing either overprovisioning or performance bottlenecks.

GPU Architecture Integrated with MIG and MPS. As shown in Figure 1, a modern GPU is built around the **streaming multiprocessor** (**SM**)–its fundamental compute block that integrates multiple **floating-point units** (**FPUs**), tensor cores, integer ALUs, and other functional units. On

Table 1. Available MIG Partitions for A100 (80GB) GPU

Profile Name	Compute	Memory	Available SM	Count
7g.80gb	7 GPC	80 GB	98	1
4g.40gb	4 GPC	40 GB	56	1
3g.40gb	3 GPC	40 GB	42	2
2g.20gb	2 GPC	20 GB	28	3
1g.10gb	1 GPC	10 GB	14	7

Notation: "xg.ygb" indicates the instance is allocated x GPCs of compute resources and y GB GPU device memory (e.g., 7g.80gb).

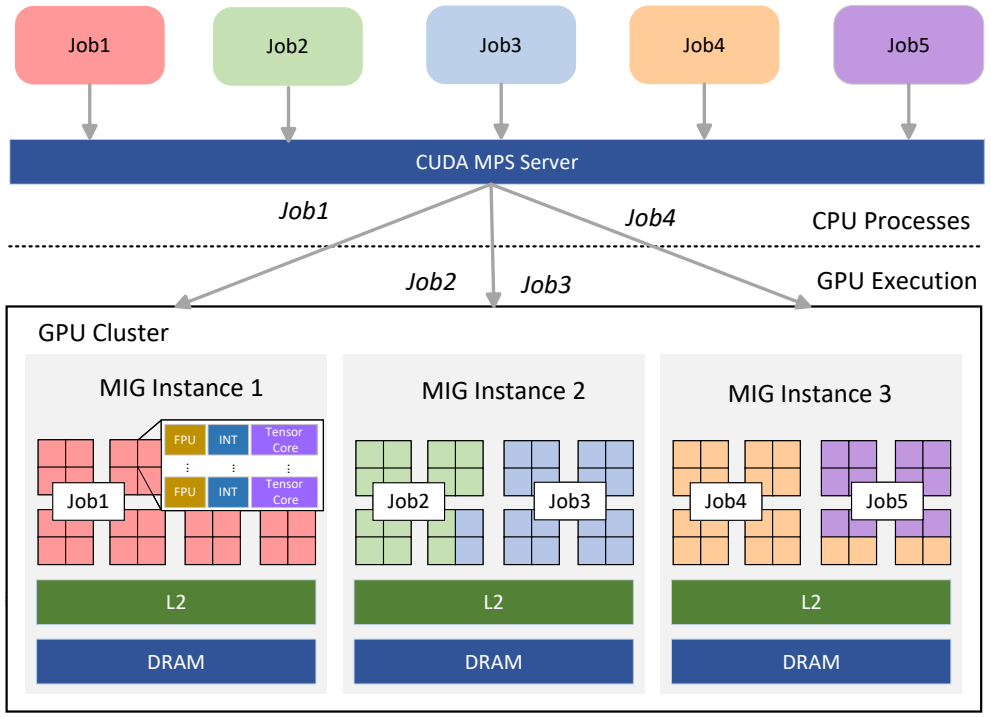

Fig. 1. Simplified GPU architecture with MIG and MPS.

top of this hardware foundation, MIG partitions the device into several isolated sub-GPUs, each owning a fixed slice of SMs and memory resources, while MPS overlays a software scheduler that multiplexes multiple processes within every sub-GPU and can dynamically reallocate the assigned SMs to follow workload demand. Therefore, the joint use of MIG and MPS balances strict isolation with elastic sharing, enabling high resource utilization across diverse DNN workloads.

2.2 Observations on Existing Technologies

Observation I

Designing an effective scheduling strategy that integrates MIG and MPS is complex, since it needs to consider the coupled configuration spaces of MIG partitions and MPS quotas when seeking configurations that maximize multi-job parallel performance.

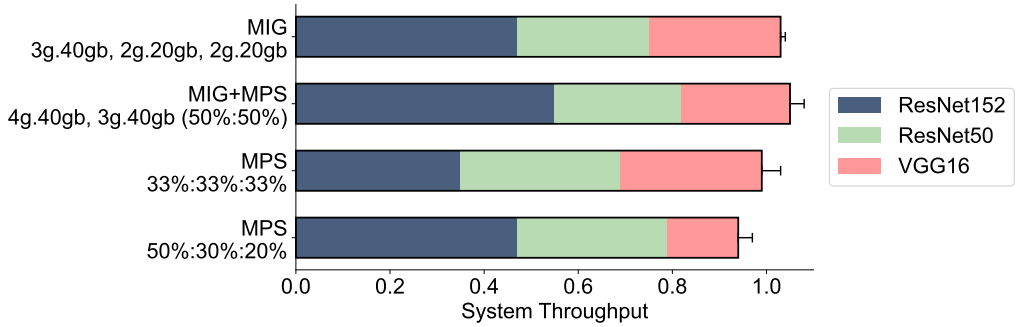

Fig. 2. System throughput of different sharing strategies. Data are presented as mean values of 20 independent runs, with upper error bars denoting +1 standard deviation.

When integrating MIG with MPS for concurrent execution, MIG provides hardware-level spatial partitioning by configuring GPU partitions, whereas MPS enables concurrent sharing within each partition through software-level configuration. 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 [10], ResNet50 [10], and VGG16 [25]–under different resource allocation strategies. As shown in Figure 2, the four evaluated configurations exhibit differences in system throughput. The configuration [4g.40gb, 3g.40gb] outperforms [3g.40gb, 2g.20gb, 2g.20gb], as the former allows ResNet50 and VGG16 to share resources more efficiently, thereby achieving higher system throughput. Although the latter also provides isolation, its smaller partitions cannot provide sufficient compute resources, resulting in lower system throughput. 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 II

Although combining MIG with MPS enables multi-job execution within partitions, the inflexibility of MIG configurations often leads to resource imbalance, where large partitions waste GPU capacity and small partitions provide insufficient resources, resulting in either overprovisioning or underprovisioning.

Although combining MIG with MPS facilitates multi-job parallelism within partitions, it still results in resource over-provisioning or under-provisioning due to fixed partition sizes. To illustrate this issue, we concurrently run three representative workloads, VGG16 [25], ResNet50 [10], and MobileNet [11], on different MIG partitions. Figure 3 shows SM utilization (left y-axis) and total execution time (right y-axis) across four configurations. In the 7g.80gb partition, all three jobs are allocated ample resources and obtain the largest amount of SM compute resources, but approximately 30% of GPU resources remain idle. When the partition size is reduced to 4g.40gb, idle SMs decrease and overall SM utilization improves. However, this higher utilization comes at the cost of performance: the reduced compute capacity in the 4g.40gb partition creates a bottleneck, leading to intensified contention and a subsequent increase in total execution time compared with the 7g.80gb case. As the partition size continues to shrink, the available resources in each partition become insufficient to support multiple jobs concurrently. The 3g.40gb partition can accommodate two jobs running in parallel, while the 2g.20gb partition can only support the execution of a single job.

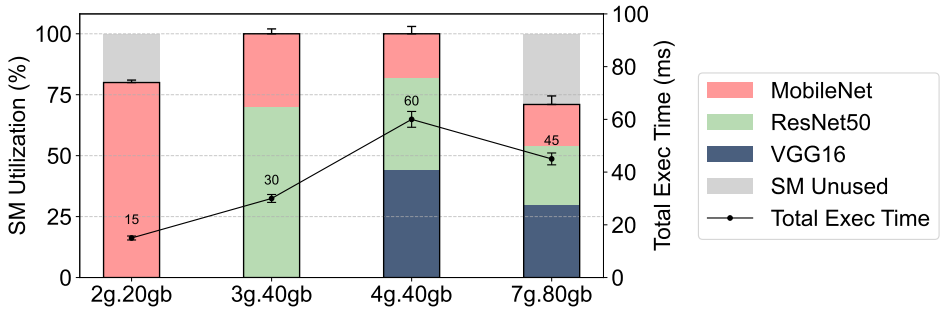

Fig. 3. SM utilization under different MIG partitions. Error bars indicate +1 standard deviation for SM utilization and ±1 standard deviation for Total Exec Time, over 20 repetitions.

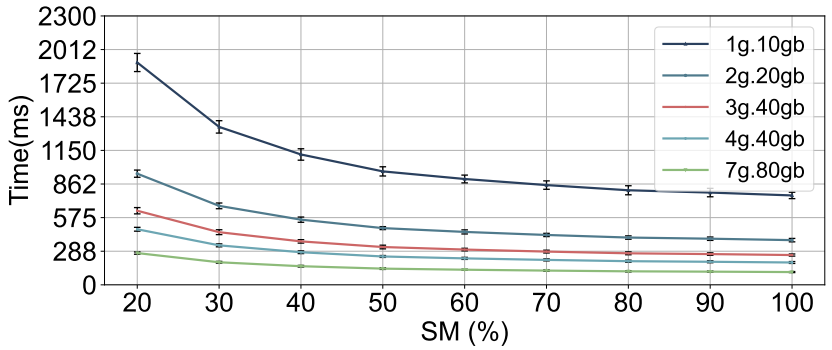

Fig. 4. SM utilization under different MPS limitations. The results are averaged over 20 trials, with error bars representing ±1 standard deviation.

Observation III

Across different MIG partitions, the degree of job slowdown under varying MPS configurations differs significantly, and how to effectively control such slowdowns to improve collocation performance remains a challenge.

The performance of a job is influenced not only by the SM share constrained by MPS but also by the computational resources provided by the assigned MIG partition; both factors jointly determine the degree of slowdown. To validate this observation, we take ResNet152 as an example and evaluate its execution time under varying MPS configurations (from 100% to 20%) across different MIG partitions (from 1g.10gb to 7g.80gb). As shown in Figure 4, reducing the MPS allocation leads to a significant increase in execution time, with slowdown magnitudes varying notably across partitions. For instance, in the 7g.80gb partition, the execution time increases from 140ms to 288ms, resulting in a slowdown of 2.2×; whereas in the 1g.10gb partition, it increases from 830ms to 1890ms, with a slowdown of only 2.28×. These results indicate that job performance is more sensitive to MPS configuration in resource-rich partitions, while such sensitivity saturates under limited resources.

Moreover, in the range of 70%–100% SM allocation, performance variations are relatively small compared with the 20%–50% range, suggesting that jobs are more sensitive in low-SM regions and achieve greater performance gains from incremental SM allocations there. This further implies that, in collocation scenarios, releasing SM resources in high-SM regions and reallocating them to low-SM regions can enhance overall resource efficiency.

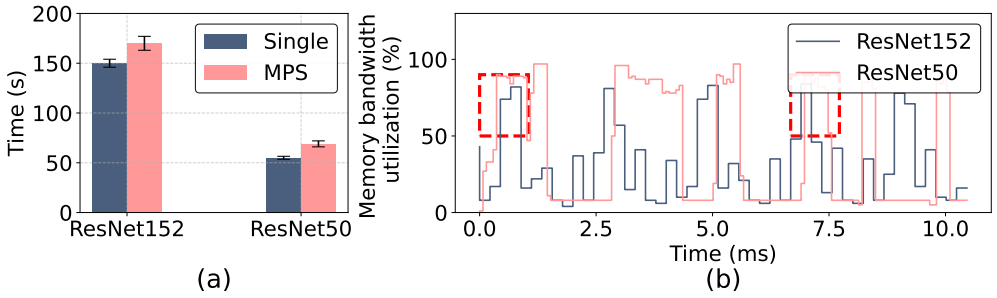

Fig. 5. SM utilization competition in a MIG partition. (a) Execution time comparison (mean ±1 s.d., n=20). (b) A representative memory bandwidth trace of co-located ResNet152 and ResNet50.

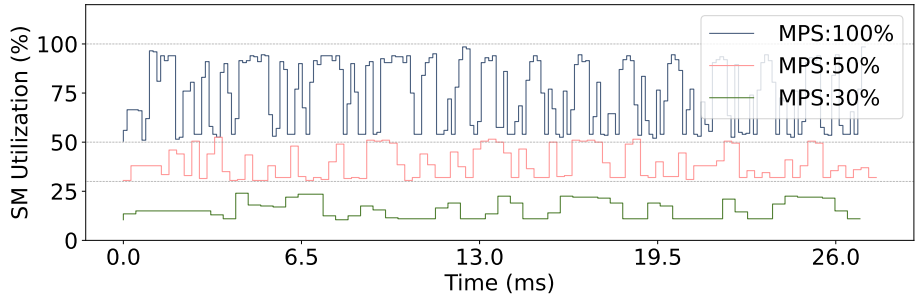

Fig. 6. SM utilization under different MPS limitations.

Observation IV

When multiple jobs are executed in parallel under MPS within the same MIG partition, resource contention remains inevitable. Even with strictly enforced allocation ratios, competition for bandwidth persists among jobs, leading to longer execution times and reduced overall efficiency.

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 5(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 with executing each job independently on the GPU with the same MPS configuration, the co-execution incurs longer execution time. This indicates that while SM utilization is constrained via MPS, performance degradation persists due to resource contention. To further investigate this, we employ Nsight Systems[3] to sequentially profile the jobs with aligned start times (Figure 5(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 Systems 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 6, with the reduction in SM allocation in MPS, kernel resource utilization decreases, and the utilization bubble grows.

[3]https://developer.nvidia.com/nsight-systems

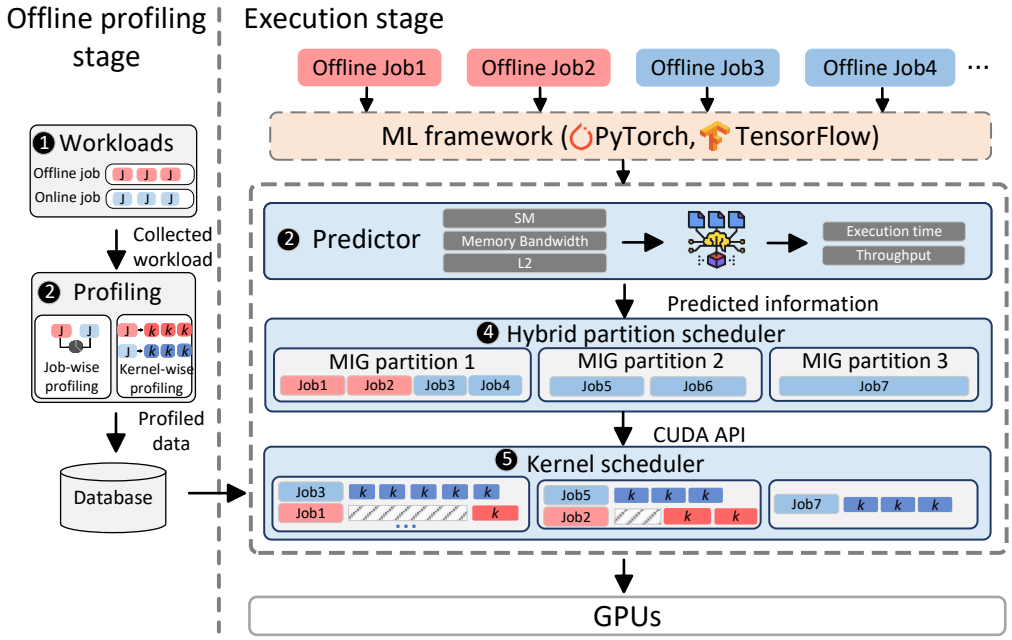

Fig. 7. System overview.

3 System Design

Figure 7 illustrates the overall architecture of **MMK**, which consists of three key 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 operates at the macro level by assigning suitable MIG partitions and configuring MPS quotas for different jobs, thereby enabling flexible and adaptive resource allocation. In contrast, the kernel scheduler operates at the micro level by intercepting CUDA APIs to perform kernel-level execution timing control and priority management, thereby alleviating fine-grained resource contention. These two components form a hierarchically coupled architecture: the physical resources allocated by the hybrid partition scheduler determine the upper bound of kernel-level management, while the kernel scheduler further reduces intra-partition interference based on the provided resource configuration, thereby significantly improving overall throughput while preserving QoS.

During the offline profiling stage, the system collects a diverse set of workloads to cover a wide range of load scenarios ❶. Both offline and online jobs are then analyzed at the job-wise and kernel-wise levels to support the performance predictor (Section 3.1), the hybrid partition scheduler (Section 3.2), and the kernel scheduler (Section 3.3) ❷. In the execution stage, the performance predictor extracts task-specific features based on resource utilization profiles ❸. The hybrid partition scheduler subsequently adjusts MIG partitions and the number of active SMs per job to maximize overall throughput ❹. Finally, the kernel scheduler applies priority-based scheduling at the kernel level to ensure the QoS of latency-sensitive online jobs ❺.

3.1 Performance Predictor

Resource contention is the fundamental cause of performance degradation when multiple jobs execute concurrently on GPUs. Ignoring contention in MIG and MPS configurations often results in severe QoS violations for latency-sensitive online jobs and significant throughput loss for offline

Table 2. Features Used to Train Predictor

Name	Description
Task type	Task's type, online job, offline job or individual kernel.
MPS configuration	MPS configuration for the task.
Task features	Task's resource features, including SM utilization, memory bandwidth utilization, and L2 cache utilization.
Kernel features	Kernel features including grid size, block size, registers per thread, and shared memory per block.
Contention features	Resource usage occupied by other collocated jobs in the same partition, including their task features and kernel features.

jobs. To address this issue, we design a performance predictor that explicitly models the interaction between job characteristics and shared resource utilization.

We adopt a **random forest** (**RF**) as the prediction model. The predictor uses five categories of input features, as summarized in Table 2. First, the task type distinguishes whether the prediction target is an online job, an offline job, or an individual kernel. This information determines both the feature collection strategy and the interpretation of the prediction output. Second, the MPS configuration specifies the fraction of SMs allocated to the task, directly reflecting the level of resource multiplexing within a partition. Third, the task features capture the intrinsic resource usage of the task, including SM utilization, memory bandwidth utilization, and L2 cache utilization, which together characterize its computational intensity and memory access behavior. Fourth, the kernel features, including grid size, block size, registers per thread, and shared memory per block, define the execution granularity and theoretical occupancy on the GPU hardware. Finally, the contention features quantify the usage of shared resources by other collocated jobs in the partition by incorporating their corresponding task and kernel features, thereby revealing the interference environment that a new job will face.

For online and offline jobs, we collect the average SM utilization, memory bandwidth utilization, and L2 cache utilization during execution, capturing their overall resource demand and behavior. To ensure accurate performance estimation, the predictor provides different outputs depending on the job type. For online jobs and kernels, the output is the predicted execution time under the current MIG/MPS configuration and contention state. For offline jobs, the output is throughput, which better reflects the system-level efficiency of long-running background workloads.

To train the model, we conduct an offline profiling phase in which workloads from Table 3 are executed under diverse MIG partitions and MPS configurations. We employ Nsight Systems to analyze execution traces, from which we extract fine-grained kernel features including grid size, block size, and register usage, alongside standard job-level metrics. In total, 2000 workload samples are generated, with 80% used for training and 20% for testing. The model is optimized with mean absolute error (MAE) as the loss function, ensuring robustness against outliers. Once trained, the predictor yields runtime performance estimates, enabling both the hybrid partition scheduler (Section 3.2) and the kernel scheduler (Section 3.3) to make informed resource allocation decisions. In Section 4.8, we evaluate the prediction accuracy of the random forest model against other machine learning techniques.

3.2 Hybrid Partition Scheduler

In this section, we propose the hybrid partition scheduler, which combines the MIG and MPS to achieve flexible resource partitioning for collocated jobs. Unlike prior approaches [22, 34], our scheduler introduces **Partition Entropy** into MIG configurations and employs **Kernel-aware**

MPS oversubscription for resource allocation. These strategies are designed to accommodate parallel execution scenarios unconstrained by fixed task limits, thereby significantly enhancing configuration efficiency and overall resource utilization.

Suppose there is a job list assigned to a GPU, which can be represented as $l = [l_1, l_2, ..., l_k]$, where k denotes the total number of jobs. 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 sub-partitions 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. We define the partition granularity g as the number of sub-partitions in a pc (i.e., $g = n$). For instance, ($[7g.80gb]$ is 1, $[4g.40gb, 3g.40gb]$ is 2, $[4g.40gb, 2g.20gb, 1g.10gb]$ is 3). 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.

Partition Entropy. To address the challenge of the expansive MIG configuration search space, we define a task-aware MIG Partition Entropy. This metric characterizes the heterogeneous adaptability between the granularity of MIG physical partitions and task resource demands, facilitating the identification of effective configurations from the perspective of partition granularity. It is defined as follows:

$$\mathcal{G}_{gran}(g) = \mathcal{H}_{integ}(g) \cdot \mathcal{B}_{sep}(g, l) \tag{1}$$

$$\mathcal{H}_{integ}(g) = \max_{pc \in pl, |pc| = g} \sum_{i=1}^{g} \left(\frac{p_i}{\sum_{i=1}^{g} p_i} \right)^2 \tag{2}$$

$$\mathcal{B}_{sep}(g, l) = \underbrace{\mathrm{Var}(\{r_j\}_{j \in l})}_{\text{Total Heterogeneity}} - \underbrace{\min_{l_1 \cup \cdots \cup l_g = l} \sum_{m=1}^{g} \mathrm{Var}(\{r_j\}_{j \in l_m})}_{\text{Residual Heterogeneity}} \tag{3}$$

Here, $\mathcal{H}_{integ}(g)$ evaluates the configuration entropy at a specific partition granularity g. By maximizing this value, the metric identifies configuration schemes with the lowest degree of physical fragmentation for a given granularity. Consequently, a higher score indicates that the system retains larger contiguous resource blocks at that granularity, reflecting stronger physical isolation capabilities and resource aggregation.

To accurately characterize resource contention among tasks, we define a multi-dimensional resource feature vector $\mathbf{u}_j$ using utilization metrics from Table 2:

$$\mathbf{u}_j = \left[u_j^{sm},\ u_j^{bw},\ u_j^{l2},\ u_j^{blk},\ u_j^{reg},\ u_j^{shm} \right], \quad r_j = \|\mathbf{u}_j\|_2 \tag{4}$$

Here, u_j^{sm}, u_j^{bw}, u_j^{l2}, u_j^{blk}, u_j^{reg}, and u_j^{shm} denote the *average* SM utilization, memory bandwidth utilization, L2 cache utilization, active-block occupancy, register utilization, and shared-memory utilization of task l_j, respectively. All metrics are collected online by the system profiler and are computed as time-averaged values over the execution of l_j, with r_j serving as a comprehensive measure of the task's overall resource intensity.

$\mathcal{B}_{sep}(g, l)$ characterizes the resource heterogeneity of the task set l at granularity g, consisting of two components: the first is the variance of the resource demands of the task set, reflecting the overall degree of resource heterogeneity; the second is the minimum sum of variances obtained by partitioning tasks into g subsets based on r_j and assigning them to g sub-partitions, which represents the degree of resource demand variation within each sub-partition. A larger difference between these two components indicates smaller variations in resource demands within each partition.

Kernel-aware MPS Oversubscription. To address persistent GPU resource fragmentation caused by imbalanced kernel-level utilization, we design a **Kernel-aware MPS oversubscription**

mechanism. In multi-job collocation scenarios, this strategy aims to maximize kernel resource utilization and enhance overall collocation efficiency. It achieves this by dynamically adjusting MPS resource limits such that the cumulative MPS allocation for the job list l exceeds 100%.We formulate the optimal MPS configuration vector m_i^{opt} for a target job l_i ($l_i \in l$) and its associated constraints as follows:

$$m_i^{opt} = m_i^{base} + \underbrace{\sigma\left(\sum_{j=1, j\neq i}^{k} \mathbf{u}_j\right)}_{\delta_i} \tag{5}$$

$$\text{s.t.} \quad \sum_{i=1}^{k} m_i^{base} = 100\% \tag{6}$$

Here, m_i^{base} denotes the baseline MPS limit for job l_i under strict resource isolation constraints. The oversubscription increment, δ_i, is defined as the standard deviation of the aggregate kernel-level resource usage of the remaining co-located jobs in list l. Specifically, we calculate the feature vector $\mathbf{u}_j$ (as defined in Equation (4)) and the statistical standard deviation of this aggregated load profile to determine δ_i. A larger δ_i indicates higher volatility (i.e., significant peaks and troughs) in the resource demands of the background job set $\{l_j | j \neq i\}$. This volatility exposes more transient resource slack, thereby allowing the target job l_i to opportunistically oversubscribe resources within these intervals.

In high-load and extreme scenarios, the kernel-aware oversubscription mechanism remains stable, because the oversubscription increment, δ_i, is bounded by the standard deviation of the aggregated kernel utilization of collocated jobs. This design smooths transient fluctuations and mitigates performance oscillation. Moreover, **MMK** first assigns each job a baseline MPS quota according to its SM utilization ratio and then applies the oversubscription increment, thereby preventing job starvation.

Optimization Objective. Our optimization objective is to maximize the overall throughput of all jobs under feasible MIG and MPS configurations. The optimization objective can be expressed as:

$$\max_{(l, ml, pc)} \quad \sum_{i=1}^{k} f_i(l_i, ml, pc) \tag{7}$$

$$\text{s.t.} \quad pc \in pl, \tag{8}$$

$$ml \in M_{\text{mps}}. \tag{9}$$

where $f_i(l_i, ml, pc)$ denotes the throughput contribution of job i under configuration (ml, pc), pl is the set of all feasible MIG partition combinations, and M_{mps} is the set of candidate MPS configurations. Based on our assumptions, the time complexity of computing the optimal MIG and MPS configuration is $O(c^k)$, where c denotes all possible MIG and MPS configurations. The complexity grows exponentially with the number of jobs, making exhaustive search impractical for large-scale workloads.

Dynamic Resource configuration algorithm. We design a heuristic mechanism that coordinates MIG configuration with **Partition Entropy** and **Kernel-aware MPS Oversubscription**, aiming to maximize GPU utilization and system throughput without compromising online QoS. The algorithm operates in *three core steps*: First, it sorts jobs based on memory demands and initializes the resource states (**Initialization**). Then, it optimizes the physical partitioning via Partition Entropy to derive a set of optimal candidates (**MIG configuration**). Finally, it performs MPS allocation for online and offline jobs under each candidate to determine the best global MIG and MPS configuration (**MPS configuration**). The detailed steps are as follows:

— **Step ❶(Initialization):** The algorithm begins by sorting the job list l in descending order based on memory demand. This step ensures that resource-intensive tasks are prioritized

ALGORITHM 1: Dynamic Resource Configuration Algorithm

Input : Job list l, partition combination pc (e.g., [4g.40gb, 3g.40gb]), partition combination list pl
Output : Optimal MIG and MPS configuration (opt_pc, opt_mps)
Initialization : Sort l by memory demand descending; $max\_stp \leftarrow 0$

1 **while** *System is running* **do**
  `// MIG configuration`
2 **if** *Conditions for reconfiguration are met* **then**
3 $min\_entropy \leftarrow$ Calculate minimum **Partition Entropy** for l;
4 $granularity \leftarrow$ Determine target granularity based on $min\_entropy$;
5 $candidates \leftarrow$ Select partitions from pl matching $granularity$;
6 **foreach** $pc \in candidates$ **do**
7 $tmp\_stp, tmp\_mps \leftarrow$ Calculate the throughput and MPS for pc using `MPSConfig`;
8 **if** $tmp\_stp > max\_stp$ **then**
9 $max\_stp \leftarrow tmp\_stp$;
10 Record optimal MIG opt_pc and MPS opt_mps;
11 Reconfigure optimal MIG opt_pc and MPS opt_mps;
  `// MPS configuration`
12 **Function** `MPSConfig`($jobs$)**:**
13 Separate $jobs$ into $jobs_{off}$ and $jobs_{on}$;
14 Sort jobs by efficiency ($\frac{\text{Throughput}}{\text{SM Utilization}}$) descending ;
15 $mps\_config \leftarrow \{\}$;
16 **foreach** *Group* $S \in \{jobs_{off}, jobs_{on}\}$ **do**
17 $total\_sm \leftarrow$ Compute the total SM utilization of S ;
18 **foreach** $job \in S$ **do**
19 $\Delta sm_i \leftarrow$ Calculate **Oversubscription** quota;
20 $mps\_config[job] \leftarrow \frac{job.sm}{total\_sm} \times limit + \Delta sm_i$;
21 $stp \leftarrow$ Estimate total throughput for $jobs_{off}$;
22 **return** stp, mps_config ;

during the initial allocation. It also initializes the maximum system throughput variable max_stp to zero to prepare for the optimization loop.

— **Step ❷(MIG configuration: Partition Entropy-driven + candidate-set pruning):** We perform MIG configuration to provision optimal hardware partitions for both online and offline jobs. To avoid disrupting latency-sensitive online jobs, *we restrict MIG reconfiguration exclusively to offline jobs, triggering it only when no online job is running in the current partition or the run queue is empty*, thereby creating a safe reconfiguration window. We first check if the system meets the conditions for reconfiguration to ensure safe transitions **(Line 2)**. The algorithm then calculates the minimum Partition Entropy to quantify the resource mismatch and determines the target partition granularity **(Lines 3–4)**. Based on this granularity, it selects a set of matching partition candidates from the global list **(Line 5)**. For each candidate, the algorithm calls `MPSConfig` to calculate the potential MPS settings and throughput. It compares these results to update the global maximum throughput and records the optimal MIG and MPS combination **(Lines 6–10)**. Finally, the system reconfigures the hardware with the identified optimal settings **(Line 11)**.

— **Step ❸(MPS configuration: Efficiency-driven sorting + Oversubscription):** Given the MIG partition selected in Step ❷, we allocate MPS resources to newly arrived online and offline jobs. To ensure that online jobs remain uninterrupted, *we restrict MPS reconfiguration exclusively to offline jobs, triggering it only upon the arrival or completion of an offline workload within a partition.* The `MPSConfig` function starts by separating jobs into online and offline groups and sorting them by computational efficiency **(Lines 13–14)**. It then iterates through each group to calculate the total SM utilization **(Lines 16–17)**. For every job in the group, the algorithm computes an oversubscription quota using Equation (5). It then assigns the optimal MPS limit by combining proportional scaling with this dynamic quota to mitigate resource contention between online and offline jobs while maximizing SM utilization **(Lines 18–20)**. The function concludes by estimating the total throughput for offline jobs and returning the optimal configuration plan **(Lines 21–22)**. Moreover, we estimate the aggregate throughput only for offline jobs. This estimate is used to guide MPS reconfiguration for offline jobs, while we do not reconfigure MPS for online jobs and thus do not account for their throughput.

Complexity Analysis: The time complexity of our algorithm dynamically adapts between $\mathcal{O}(N\log N)$ and $\mathcal{O}(N^2 \log N)$. In the best-case scenario, the system only reallocates MPS quotas without MIG reconfiguration, requiring only $\mathcal{O}(N\log N)$ for job sorting. In the worst-case scenario, significant workload fluctuations trigger a global MIG reconfiguration, where the entropy-based pruning strategy reduces the search space from exponential to $\mathcal{O}(N^2 \log N)$. In the average-case scenario, since high-overhead MIG reconfiguration is infrequent, the complexity approximates $\mathcal{O}(N\log N)$, thereby ensuring scalability in large-scale cluster scenarios.

3.3 Kernel Scheduler

As mentioned in Challenge 2, the coarse-grained resource allocation of MPS struggles to mitigate kernel-level contention among jobs. This makes it difficult to guarantee the QoS of online jobs and the performance of offline jobs in multi-workload scenarios. Therefore, we design a kernel scheduler to enable fine-grained resource scheduling.

Algorithm 2 demonstrates the core scheduling strategy of **MMK**, addressing resource contention across *three scenarios*: enabling parallel execution of online tasks under low-load or burst conditions via priority-based kernel scheduling and MPS (**Online vs. Online**), allocating execution budgets for offline jobs by calculating the slack time of online jobs while ensuring QoS (**Online vs. Offline**), and maximizing computational resource utilization by batching kernels from multiple offline jobs (**Offline vs. Offline**). The detailed procedure of Algorithm 2 is described as follows:

Online vs. Online. To arbitrate contention among online jobs, **MMK** performs slack-aware admission control **(lines 3–8)**. The scheduler first applies the MPS configuration for the online job **(line 3)** to enhance execution parallelism under burst scenarios, and subsequently computes the global minimum slack *min_slack* by invoking `GetMinSlackTime` **(line 4)**. Here, slack is defined as the QoS target minus the observed execution time of an online job **(lines 19–20)**, serving as a safety window. **MMK** then checks resource contention via `CheckContention`, which evaluates task-level and kernel-level features in Table 2, and ensures *min_slack* exceeds the job's execution time **(line 5)**. If satisfied, **MMK** launches the kernel **(line 6)**. Otherwise, it sets the online launch flag to false to prevent QoS violations **(lines 7–8)**. Moreover, when multiple online jobs arrive concurrently, they are executed in parallel via MPS, and each job's MPS setting is provided by the hybrid scheduler.

Online vs. Offline. To prioritize online jobs, **MMK** applies the offline MPS configuration **(line 9)** and computes a kernel execution budget `Thres` **(line 10)**. The threshold is derived as $\texttt{Thres} = \textit{min\_slack} \times \textit{job\_offline.mps}$. Iterating through the offline kernels **(line 11)**, a kernel is admitted to the candidate list only if there is no active online job or if the cumulative offline execution time *job_offline.thres* stays below `Thres` **(lines 12–14)**.

ALGORITHM 2: Kernel Scheduling Algorithm

```
   Input          : jobs_online, jobs_offline
   Initialization : launch_online ← True; offline_kernel_list ← ∅; min_slack ← ∞
1  while True do
2      ava_res ← GetResources(); job_online ← GetOnline(); job_offline ← GetOffline() ;
       // Part 1: Online vs. Online
3      ApplyMPSConfig(job_online) ;
4      min_slack ← GetMinSlackTime() ;
5      if CheckContention(job_online, ava_res) ∧ min_slack > job_online.exectime then
6          LaunchKernel(job_online) ;
7      else
8          launch_online ← False ;
       // Part 2: Online vs. Offline
9      ApplyMPSConfig(job_offline) ;
10     Thres ← min_slack × job_offline.mps ;
11     foreach kernel ∈ job_offline do
12         if job_online is None or job_offline.thres < Thres then
13             Append kernel to offline_kernel_list;
14             job_offline.thres ← job_offline.thres + kernel.time ;
       // Part 3: Offline vs. Offline
15     if offline_kernel_list ≠ ∅ then
16         Split offline_kernel_list into packages P s.t. usage ≤ 100% ;
17         foreach p ∈ P do
18             LaunchKernel(p) ;
19 Function GetMinSlackTime:
20     return min{j.qos − j.exec | j ∈ GetRunningOnlineJobs()} ;
```

Offline vs. Offline. Since offline jobs have no strict QoS constraints, **MMK** focuses on maximizing throughput. It checks if the collected `offline_kernel_list` is non-empty **(line 15)**. If so, the scheduler bundles kernels from concurrent offline jobs into multiple kernel packages P by leveraging task-level and kernel-level features in Table 2, such that each package stays within the partition resource budget (e.g., usage $\leq$ 100%), and then launches execution at the package granularity **(lines 16–18)**.

As shown in Figure 8, when the GPU hosts two online jobs A and B along with one offline job C, the scheduler prioritizes the kernels of A and B under resource contention, while C is suspended. **MMK** then invokes `getslacktime` to obtain the minimum slack time of all online jobs in the current partition based on *slack time* 1 and *slack time* 2, and computes the execution threshold *Thres* for C. After executing kernels $k1$ through $k3$, C enters a waiting state because it has exhausted its allowed execution time until A and B finish. Once A and B complete, C resumes if no new online job arrives; otherwise, upon the arrival of a new online job, the scheduler recomputes *Thres* for C and proceeds accordingly.

4 Performance Evaluation

4.1 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.

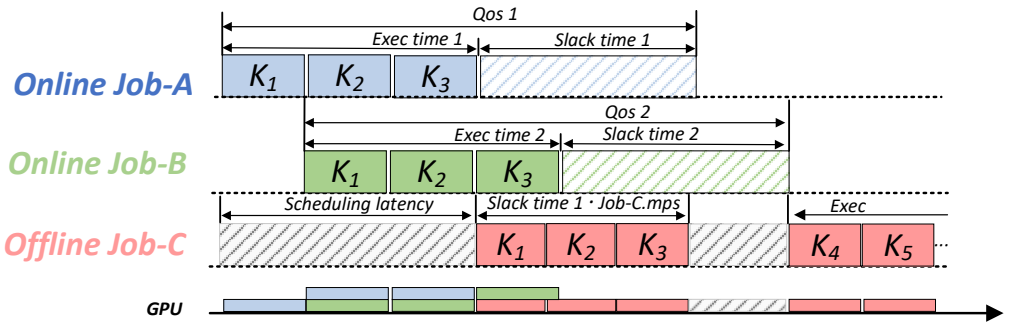

Fig. 8. The kernel scheduling process.

For hybrid partition scheduling, **MMK** leverages both MIG and MPS APIs. The hybrid scheduler relies on MIG and MPS. MIG depends on NVIDIA GPUs that support MIG, which are available starting from the Ampere generation. MPS relies on NVIDIA's CUDA software stack and requires GPUs with compute capability no lower than 3.5. 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, collecting runtime metrics such as SM utilization, memory bandwidth, L2 utilization and memory consumption. These metrics are used to predict job priorities and control the timing of API releases accordingly. By contrast, the API interception technique used by the kernel scheduler is built on the Linux dynamic linking mechanism rather than any specific CUDA feature.

4.2 Experimental Setup

Evaluation Setup: The experimental setup includes two Intel Xeon Platinum 8369B CPU with 64 threads and two NVIDIA A100-PCIe 80GB GPUs. **MMK** runs on Ubuntu 20.04 with Linux kernel 5.15. We deploy NVIDIA Driver 535.154.05 and CUDA 12.2, which provide the foundational support for MPS and MIG. For deep learning workloads, our primary evaluation is based on PyTorch 2.4.1, linked against cuDNN 8.9.7 and cuBLAS 12.2 to achieve optimized operator execution. Detailed performance profiling and kernel-level analysis are conducted using Nsight Systems 2023.2.3 and Nsight Compute 2023.2.2. Regarding implementation mechanics, MMK intercepts CUDA APIs via the `LD_PRELOAD` mechanism, functioning as a transparent shim layer that is effectively decoupled from the deep learning frameworks. To simultaneously orchestrate MIG, MPS, and kernel-level scheduling, MMK is designed with broad compatibility, supporting CUDA versions from 11.6 to the latest 12.x series. Furthermore, it is compatible with mainstream PyTorch versions ranging from 1.13 to 2.4.1, requiring no modifications to the framework source code.

Workloads: We follow the workloads settings of MISO[22] and generate a job trace based on Helios[12]. For the testbed evaluations, we create 60 online and 40 offline jobs from Table 3. The job arrival intervals follow a Poisson distribution with λ = 30 seconds. For the simulation experiments, we configure 650 online and 350 offline jobs. Following the methodology of [34], we design an event-driven simulator to conduct the large-scale multi-GPU simulation, where system state is updated only upon job arrivals, completions, and reconfiguration events, enabling efficient evaluation of scheduling strategies at scale.

Table 3. Workloads Used to Evaluate **MMK**

Name	Workload	Batch size	QoS (ms)
VGG16[25]	Inference	8, 16, 32, 64, 128	170
MobileNet[11]	Inference	8, 16, 32, 64, 128	40
ResNet18[10]	Inference	8, 16, 32, 64, 128	35
BERT[19]	Train	2, 4, 8, 16	-
ResNet152[10]	Train	16, 32, 64, 128	-
DenseNet201[15]	Train	16, 32, 64, 128	-
Transformer[29]	Train	16, 32, 64, 128	-
DenseNet121[15]	Inference and train	16, 32, 64, 128	90
VGG19[25]	Inference and train	16, 32, 64, 128	200
ResNet50[10]	Inference and train	16,32, 64, 128	80
Qwen2-7B [28]	Inference and train	1	50
Llama-3.2-3B [6]	Inference and train	1	40
DeepSeek-R1-1.5B [7]	Inference and train	1	35

We apply the following metrics to evaluate performance of **MMK**.

— **Average Job Completion Time (JCT)**: Average JCT refers to the time it takes for a job to complete from the start of execution, with a smaller value indicating better system efficiency and faster job completion.

— **System Throughput (STP)**. Following MISO [22], we define STP as the combined progress rate of all co-located jobs. This metric quantifies the overall processing rate by comparing the execution speed of jobs in a co-located environment against their standalone baselines. Formally, for a set of m jobs, let p_i denote the standalone execution speed (e.g., mini-batches per second) of job i when running exclusively on the GPU, and let q_i denote its current execution speed under co-location. The System Throughput is calculated as:

$$\text{STP} = \sum_{i=1}^{m} \frac{q_i}{p_i} \tag{10}$$

By aggregating these normalized speeds, STP reflects the effective system-wide throughput improvement relative to sequential execution.

— **Makespan**: Makespan refers to the completion time of the job with the longest completion time among all jobs.

Baselines: We compare **MMK** with four baselines.

— **Default**: In the default pattern, all the workloads are scheduled to the GPU without consideration of job interference. It uses time-sharing to provide the same GPU resources for each job without considering the impact of resource competition between jobs.

— **MPS**: We adopt the official MPS-based GPU sharing scheme, in which all workloads share the same GPU context to minimize context switching overhead. This approach enhances the parallel execution capability of GPUs and improves resource utilization, although it provides limited resource isolation.

— **MISO** [22]: We follow the MISO's methodology to implement its framework. When the number of workloads is fewer than the maximum number of partitions, we allocate each workload with the minimum average job completion time; otherwise, we follow the **first-come-first-served** (**FCFS**) scheduling pattern, and the extra workloads have to wait for idle resources.

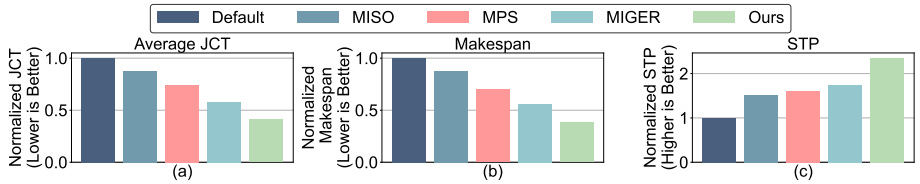

Fig. 9. Performance comparison between competing strategies. All results are normalized to default.

— **MIGER** [34]: This method integrates MIG and MPS, placing at most one offline job and one online job within each partition, and assigning the optimal MPS configuration to each job. When a more optimal MIG and MPS configuration becomes available, the method performs reconfiguration.

4.3 Performance Comparison

As shown in Figure 9, we compare the performance of five scheduling strategies in terms of average JCT, makespan, and STP. For average JCT, **MMK** reduces the value by 53.1%, 44.5%, and 28% compared with MISO, MPS, and MIGER, respectively. MISO suffers from high queuing delays due to the absence of intra-partition parallelism; MPS enables concurrency but lacks hardware-level resource isolation, making it prone to bandwidth and L2 cache contention; MIGER combines MIG and MPS to alleviate interference but still relies on coarse-grained control within partitions. In contrast, **MMK** maintains MIG-based isolation and elastic MPS sharing while introducing a kernel-level priority scheduling mechanism that prioritizes online job kernels upon contention and slices offline jobs into controllable execution segments, thereby reducing kernel queuing delays and mitigating long-tail execution. For instance, in concurrent execution of online ResNet50 inference and offline ResNet152 training, this mechanism effectively suppresses bandwidth and L2 cache contention, significantly reducing the waiting time of latency-sensitive kernels.

For makespan, **MMK** achieves reductions of 56.7%, 45.7%, and 32% over MISO, MPS, and MIGER, respectively. This advantage stems not only from a more balanced MIG–MPS resource orchestration strategy but also from the ability of kernel-level scheduling to suppress long-tail jobs. Upon detecting intra-partition resource contention, **MMK** dynamically adjusts the launch timing and execution duration of offline kernels, preventing prolonged occupation of critical resources and mitigating the delay impact of long-tail stages in the overall completion time.

For STP, **MMK** improves performance by 54.3%, 45.9%, and 35% over MISO, MPS, and MIGER, respectively. Guided by performance prediction, **MMK** allocates larger SM shares to jobs in performance-sensitive phases, while the kernel-level priority scheduler preserves online job performance at the granularity of individual kernels under contention and opportunistically executes offline kernels during idle periods. These mechanisms jointly enhance SM utilization and computation overlap. Collectively, they enable **MMK** to consistently outperform all baseline methods across the three metrics, with performance gains directly attributable to fine-grained kernel-level scheduling that mitigates resource conflicts within partitions and rigorously enforces QoS for online jobs.

As shown in the **cumulative distribution function** (**CDF**) curve of relative JCT in Figure 10, compared with other baseline methods, **MMK** exhibits a steeper rise in the low-latency region and converges earlier at the tail. This indicates that it effectively reduces the queuing delay for short jobs and significantly improves the execution efficiency of large, long-running jobs. This performance advantage is primarily attributed to the synergy of two mechanisms: First, **MMK** dynamically optimizes MIG and MPS configurations based on the resource requirements and execution characteristics of the jobs, ensuring resource isolation while increasing intra-partition concurrency and

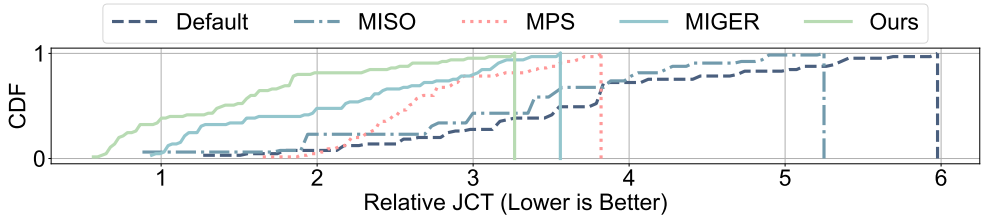

Fig. 10. CDF comparison of JCTs.

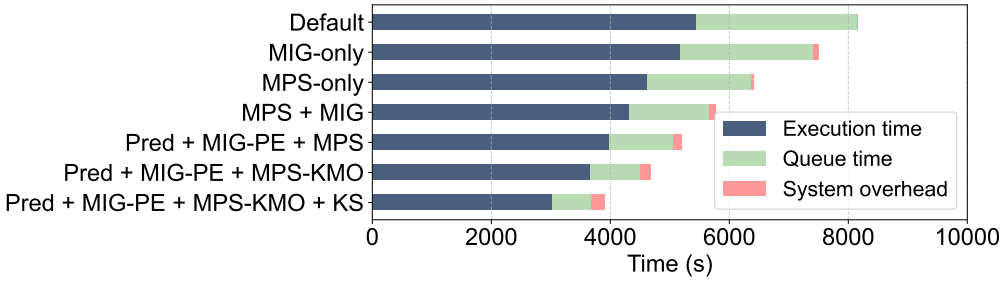

Fig. 11. Performance breakdown analysis of **MMK**. Here, Pred stands for Predictor, MIG-PE denotes MIG with partition entropy, MPS-KMO represents MPS with Kernel-aware MPS Oversubscription, and KS indicates Kernel Scheduling.

avoiding resource underutilization caused by static allocation. Second, the fine-grained kernel-level scheduling strategy prioritizes the scheduling of short jobs upon detecting resource contention, thereby reducing the waiting time of critical kernels and effectively suppressing long-tail delays.

4.4 Performance Analysis Breakdown

We conduct a performance breakdown analysis of the key components of **MMK**, as shown in Figure 11. Using a fixed 7g.80gb MIG partition without reconfiguration as the baseline, we observe that the lack of intra-partition parallelism and cross-job isolation results in the highest execution time and queuing delay. When multiple jobs arrive simultaneously, high–resource-demand jobs occupy computing resources for extended periods, substantially increasing the waiting time of subsequent jobs.

While MIG-only provides hardware isolation, its static partitioning still causes resource fragmentation and severe queuing delays; conversely, MPS-only alleviates queuing pressure but remains limited by cross-task interference. Combining the two in MPS + MIG improves both execution efficiency and waiting time, yet it still relies on plain MIG and MPS without the optimized mechanisms of **MMK**. With predictor-guided MIG-PE and MPS (Pred + MIG-PE + MPS), the system makes more informed partition decisions, reducing execution time and queuing delay by 26.8% and 59.9%, respectively, compared with the baseline. Replacing plain MPS with MPS-KMO (Pred + MIG-PE + MPS-KMO) further improves these reductions to 32.7% and 68.8%, demonstrating that Kernel-aware MPS Oversubscription more effectively exploits transient slack and mitigates intra-partition contention.

Finally, integrating KS on top of Pred + MIG-PE + MPS-KMO (Pred + MIG-PE + MPS-KMO + KS) enables kernel-granularity resource scheduling, prioritizing online-job kernels while deferring or fragmenting offline-job kernels. The full design reduces execution time and queuing delay by 44.4% and 75.8%, respectively, compared with the baseline, and by a further 17.4% and 22.5% over Pred +

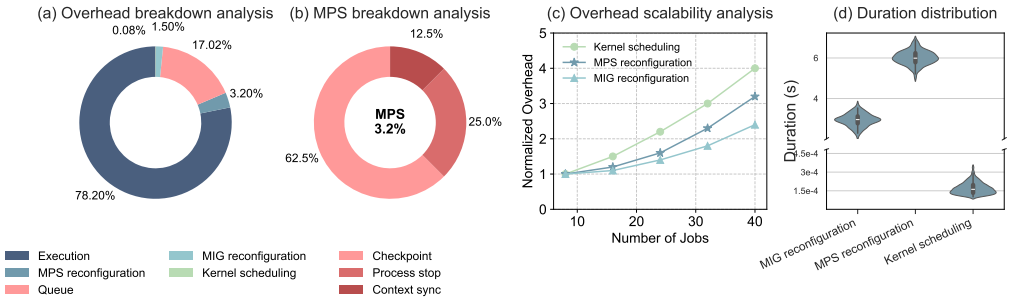

Fig. 12. System overhead analysis.

MIG-PE + MPS-KMO. Although these optimizations introduce additional control overhead, the gains in execution and queuing time dominate, confirming that the predictor, MIG-PE, MPS-KMO, and KS provide complementary benefits.

4.5 System Overhead

In this subsection, we present a breakdown analysis of the system overhead introduced by **MMK**, focusing on three core components: MIG reconfiguration, MPS reconfiguration, and fine-grained kernel-level scheduling.

MIG Reconfiguration Overhead. As illustrated in Figure 12(a), MIG reconfiguration overhead accounts for only 1.5% of the total workload lifecycle, representing a relatively minor fraction of the overall system overhead. Although the physical duration of a single MIG reconfiguration operation is on the order of seconds (Figure 12(d)), its cumulative proportion during long-running periods remains low. This is primarily attributed to the hybrid scheduler, which maximizes job residency density within single partitions, thereby significantly reducing the frequency of reconfiguration. Furthermore, as shown in Figure 12(c), the overhead exhibits a moderate growth trend as the job scale increases, verifying the scalability of the system in multi-task scenarios.

MPS Reconfiguration Overhead. MPS reconfiguration introduces a 3.2% system overhead, as shown in Figure 12(a), which is primarily incurred by the reconfiguration of offline jobs. As detailed in Figure 12(b), this overhead consists of Checkpoint (62.5%), Process stop (25.0%), and Context sync (12.5%). Specifically, offline jobs require state preservation via checkpointing prior to reconfiguration, followed by process stop. Subsequently, context sync overhead arises when restarting the offline job processes under the new MPS configuration. Figure 12(c) further reveals the trend of MPS reconfiguration overhead under varying loads; the overhead maintains a flat trend despite a significant increase in the number of concurrent jobs, confirming scalability in multi-task scenarios. Additionally, the duration distribution in Figure 12(d) indicates that MPS reconfiguration incurs seconds-level time costs, reflecting the inherent physical cost of these operations.

Kernel Scheduling Overhead. To enhance GPU utilization while prioritizing the QoS of online jobs, **MMK** implements fine-grained kernel-level scheduling. As shown in Figure 12(a), the average scheduling overhead of this mechanism accounts for merely 0.08% of the total execution time. In contrast to the seconds-level overheads of MIG and MPS, kernel scheduling operates at the microsecond level (Figure 12(d)). This low-latency characteristic ensures high efficiency in task switching and priority determination. Furthermore, Figure 12(c) illustrates the scalability of our approach. As the number of concurrent jobs increases from 10 to 40, the kernel scheduling overhead remains consistently stable and negligible compared with reconfiguration overheads, demonstrating that **MMK** effectively supports large-scale multi-job scenarios without incurring significant scheduling penalties.

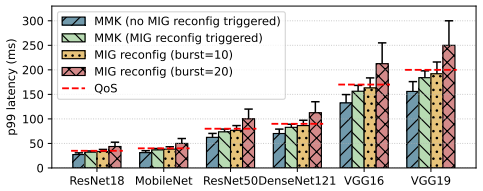

Fig. 13. Performance analysis of MIG reconfiguration.

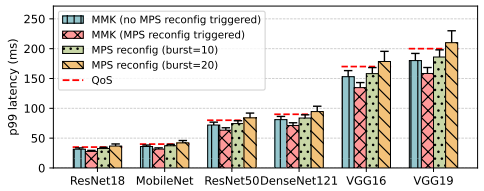

Fig. 14. Performance analysis of MPS reconfiguration.

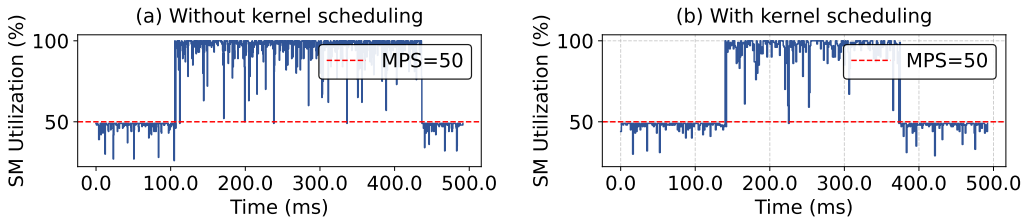

Fig. 15. SM utilization comparison between with vs. without kernel scheduling.

4.6 Effectiveness Analysis

Effectiveness analysis of MIG Reconfiguration. To evaluate the impact of MIG reconfiguration overhead on the QoS of online tasks, Figure 13 illustrates the p99 latency distribution of online inference tasks under various burst scenarios. We compare the performance of MMK during MIG reconfiguration against a stable state (without reconfiguration) and high-concurrency burst scenarios (Burst=10 and Burst=20). Since MIG reconfiguration occurs exclusively in partitions free of online jobs, it restricts the availability of partitions for online tasks, resulting in a 17.9% increase in average p99 latency compared with the state without reconfiguration. However, MMK effectively maintains the QoS of online tasks via its fine-grained kernel scheduler. Even in the Burst=10 scenario, only 5% of online tasks experience QoS violations. Significant latency degradation is observed only under the extreme load surge of Burst=20, primarily due to intensified resource contention. These results demonstrate that MMK effectively mitigates performance jitter induced by dynamic reconfiguration, ensuring the reliable execution of online tasks.

Effectiveness analysis of MPS Reconfiguration. Figure 14 evaluates the impact of MPS reconfiguration overhead on the QoS of online tasks, presenting the p99 latency of online inference tasks under various burst scenarios. Notably, the average p99 latency during MPS reconfiguration decreased by 12.0% compared with the state without reconfiguration. This performance gain is primarily attributed to the fact that MPS reconfiguration exclusively targets offline jobs, suspending their execution during the process. Consequently, online tasks acquire more computational resources, leading to reduced latency. Similarly, in high-concurrency burst scenarios (Burst=10 and Burst=20), the elimination of contention from offline jobs prevents significant latency degradation, maintaining a QoS violation rate of less than 8%. These results demonstrate that MMK leverages the isolation window created during dynamic adjustment to effectively suppress interference, thereby strictly guaranteeing the real-time requirements of online tasks.

Effectiveness analysis of kernel scheduling. Figure 15(a) and Figure 15(b) compare the performance of **MMK** with and without kernel-level scheduling under identical MPS configurations. Specifically, the MPS setting for the ResNet152 training job is configured to 50% and 100% in two separate cases, while the ResNet50 inference job consistently uses 100% MPS. The results show that,

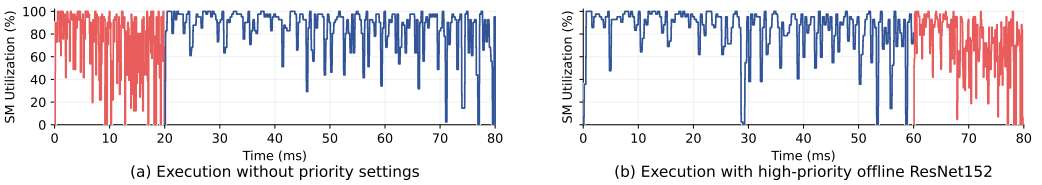

Fig. 16. Performance analysis of kernel priority inversion.

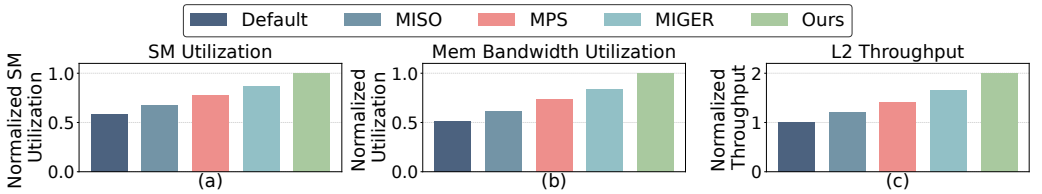

Fig. 17. Performance analysis of low-level hardware resource utilization.

in both configurations, **MMK** with kernel-level scheduling achieves a 100% increase in the execution speed of the ResNet50 inference job. This improvement is primarily attributed to the priority-based kernel scheduling policy. When resource contention between ResNet152 and ResNet50 is detected, **MMK** suspends the execution of ResNet152 kernels and reallocates the computation resources to ResNet50. This ensures that ResNet50 can fully utilize the available computational resources rather than sharing them with ResNet152, thereby significantly reducing the inference latency.

Effectiveness analysis of kernel inversion mitigation. To verify that the MMK kernel scheduler can effectively prevent priority inversion and guarantee online task QoS even under varying CUDA stream priority configurations, we conducted a sensitivity analysis. ResNet50 (online) and ResNet152 (offline) were selected as colocated workloads. Figure 16(a) shows the baseline scenario where the online task is prioritized as expected, while Figure 16(b) illustrates a stress test where the offline task's priority was artificially raised. Although ResNet152 initially preempted GPU resources (0–25ms) due to its higher priority, MMK predicted a potential QoS violation at 25ms; consequently, it forcibly suspended the high-priority task and immediately dispatched ResNet50. These results demonstrate that MMK can effectively override static stream priority settings to strictly guarantee QoS when resource contention jeopardizes online performance.

Effectiveness analysis of hardware resource utilization. To further elucidate the system's effectiveness at the hardware level, Figure 17 illustrates the underlying architectural metrics during task co-location. Compared with the baseline methods, **MMK** maintains high and stable SM utilization while significantly mitigating memory bandwidth saturation and L2 cache contention. Through efficient resource allocation and kernel scheduling, we effectively alleviate simultaneous burst accesses to the memory subsystem, thereby guaranteeing the QoS of online jobs while maintaining the performance of offline jobs, ultimately achieving optimal utilization of hardware resources.

4.7 Performance Analysis on LLM Workloads

To verify the performance of MMK in bandwidth-sensitive LLM scenarios, we extend the experiments to three different LLM scenarios (DeepSeek-R1-1.5B, Llama-3.2-3B, and Qwen2-7B) with Batch Size=1. These are co-located as online inference workloads with offline LLM training tasks to evaluate the **Time to First Token** (**TTFT**). As shown in Figure 18, baseline schemes such as MISO, MPS, and MIGER all exhibit QoS violations, while MMK maintains tail latency within the QoS threshold. Compared with Default, MPS, and MIGER, MMK reduces the average TTFT by

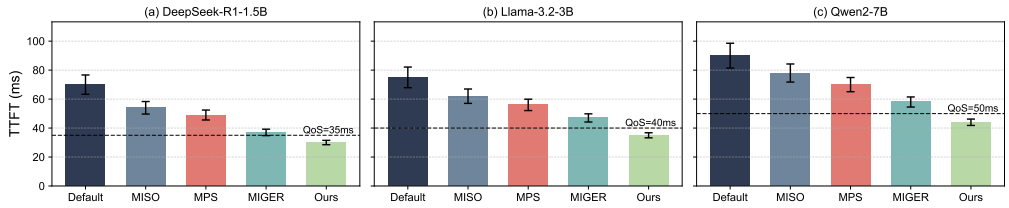

Fig. 18. Performance analysis of LLM workloads.

Table 4. 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%

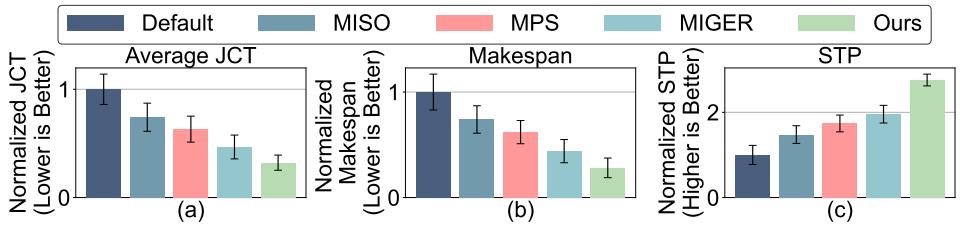

Fig. 19. Performance comparison in the simulation.

approximately 53%, 38%, and 23%, respectively. Although existing MPS and MIGER achieve spatial isolation of computational resources, they have limitations in handling bandwidth contention. In contrast, MMK mitigates long-tail latency caused by bandwidth congestion by prioritizing online inference kernels at a kernel-level granularity, thereby meeting the QoS requirements of LLM inference while maintaining system throughput.

4.8 Performance Prediction Accuracy

We choose three representative algorithms in machine learning for comparison: **linear regression (LR)**,[4] **multilayer perceptron (MLP)**,[5] and **support vector regression (SVR)**.[6] 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 4. 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.

4.9 Simulation Evaluation

As shown in Figure 19, **MMK** outperforms MIGER by 31.11%, 36.36%, and 41.18% in terms of average JCT, makespan, and STP, respectively, owing to its fine-grained isolation of offline jobs and

[4]https://scikit-learn.org/stable/api/sklearn.linear_model.html
[5]https://scikit-learn.org/stable/api/sklearn.neural_network.html
[6]https://scikit-learn.org/stable/api/sklearn.svm.html

kernel-level priority scheduling. While MIGER achieves better performance than MISO by enabling intra-partition concurrency and partially mitigating SM contention, its intra-partition resource control remains relatively coarse-grained, making it difficult to prevent short-job queuing delays under high-concurrency workloads. MISO, in contrast, completely lacks intra-partition parallelism, resulting in even higher queuing delays. Under such conditions, the Default baseline suffers the most due to severe resource contention. In comparison, **MMK** maintains robust performance even at simulation scale by increasing concurrency across multiple GPUs and partitions, alleviating resource contention, thereby simultaneously improving both latency and throughput for large-scale workloads.

5 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. AntMan [31] co-designs cluster schedulers with deep learning frameworks, enabling fine-grained coordination of memory and computation for efficient multi-job execution. TGS [30] provides transparent GPU sharing with adaptive rate control and unified memory, ensuring high utilization and performance isolation. SMSS [4] proposes a serverless-based model serving framework that improves GPU time sharing to reduce cold-start overhead and enhance utilization. SMORE [23] proposes a serverless-based co-location scheduling framework that optimizes GPU time-sharing by predicting performance degradation. However, existing methods fail to fully utilize GPU resources in multi-job concurrent scenarios due to the lack of consideration for spatial sharing. In contrast, we maximize GPU utilization by designing an efficient GPU sharing strategy.

Spatial sharing. MIGER [34] combines MPS and MIG to dynamically determine partition sizes and allocate MPS resources based on task requirements. MIG-serving [27] uses Genetic Algorithms and Monte Carlo Tree Search for efficient GPU partitioning in DNN inference, minimizing resource contention. Paris [20] combines partitioning and elastic scheduling for multi-GPU inference. Despite these advancements, MIG faces challenges in agility due to long partition creation times. MPS, utilized in Ekya [2] and INFless [33], enables parallel execution of inference tasks but suffers from high interference in shared resources, leading to suboptimal performance. Fine-grained kernel scheduling, as seen in REEF [8] and Orion [26], reduces interference and improves resource utilization during spatial sharing, but still faces tradeoffs between isolation and efficiency. ParvaGPU [21] combines MIG for spatial partitioning with MPS for fine-grained concurrency control, enabling efficient spatial GPU sharing that alleviates underutilization and fragmentation while ensuring diverse SLO compliance in large-scale DNN inference. Previous methods either focus solely on MIG or MPS, or combine the two, but they overlook fine-grained kernel scheduling under these mechanisms. We efficiently integrate MIG, MPS, and kernel-level scheduling to achieve fine-grained resource management and performance guarantees.

6 Conclusions

In this article, 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 the state-of-the-art method, **MMK** achieves 28% lower average job completion time, 32% lower makespan, and 35% higher system throughput.

References

[1] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav S. Gulavani, Ramachandran Ramjee, and Alexey Tumanov. 2024. Vidur: A large-scale simulation framework for llm inference. In *Proceedings of Machine Learning and Systems*.

[2] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. 2022. Ekya: Continuous learning of video analytics models on edge compute servers. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems* 33 (2020), 1877–1901.

[4] Zinuo Cai, Zebin Chen, Ruhui Ma, and Haibing Guan. 2023. SMSS: Stateful model serving in metaverse with serverless computing and GPU sharing. *IEEE Journal on Selected Areas in Communications* 42, 3 (2023), 799–811.

[5] Li Chen, Penghao Wu, Kashyap Chitta, Bernhard Jaeger, Andreas Geiger, and Hongyang Li. 2024. End-to-end autonomous driving: Challenges and frontiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024).

[6] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv e-prints* (2024), arXiv–2407.

[7] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[8] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*.

[9] Xueyuan Han, Zinuo Cai, Yichu Zhang, Chongxin Fan, Junhan Liu, Ruhui Ma, and Rajkumar Buyya. 2024. Hermes: Memory-efficient pipeline inference for large models on edge devices. In *Proceedings of International Conference on Computer Design*. IEEE.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*.

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

[12] Qinghao Hu, Peng Sun, Shengen Yan, Yonggang Wen, and Tianwei Zhang. 2021. Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*.

[13] Qinghao Hu, Zhisheng Ye, Zerui Wang, Guoteng Wang, Meng Zhang, Qiaoling Chen, Peng Sun, Dahua Lin, Xiaolin Wang, Yingwei Luo, et al. 2024. Characterization of large language model development in the datacenter. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*.

[14] Qinghao Hu, Zhisheng Ye, Meng Zhang, Qiaoling Chen, Peng Sun, Yonggang Wen, and Tianwei Zhang. 2023. Hydro: Surrogate-Based hyperparameter tuning service in datacenters. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*.

[15] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. 2017. Densely connected convolutional networks. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. 4700–4708.

[16] Peiwen Jiang, Haoxin Wang, Zinuo Cai, Lintao Gao, Weishan Zhang, Ruhui Ma, and Xiaokang Zhou. 2024. Slob: Suboptimal load balancing scheduling in local heterogeneous gpu clusters for large language model inference. *IEEE Transactions on Computational Social Systems*. 7941–7951

[17] Zhuolong Jiang, Zinuo Cai, Hongyu Zhao, Baoheng Zhang, Tianqi Wu, Yiming Qiang, Ruhui Ma, Haibing Guan, and Rajkumar Buyya. 2025. HeShare: Energy-aware and efficient multi-task GPU sharing in heterogeneous GPU-based computing systems. *IEEE Trans. Comput.*. 776–787.

[18] Lingxiao Jin, Zinuo Cai, Haoxin Wang, Zongpu Zhang, Ruhui Ma, Haibing Guan, and Yuan Liu. 2025. Ephemera: Accelerating I/O-intensive serverless workloads with a harvested in-memory file system. *ACM Transactions on Architecture and Code Optimization* 22, 3 (2025), 1–24.

[19] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of Association for Computational Linguistics: Human Language Technologies*.

[20] Yunseong Kim, Yujeong Choi, and Minsoo Rhu. 2022. Paris and elsa: An elastic scheduling algorithm for reconfigurable multi-gpu inference servers. In *Proceedings of Design Automation Conference*.

[21] Munkyu Lee, Sihoon Seong, Minki Kang, Jihyuk Lee, Gap-Joo Na, In-Geol Chun, Dimitrios Nikolopoulos, and Cheol-Ho Hong. 2024. ParvaGPU: Efficient spatial GPU sharing for large-scale DNN inference in cloud environments. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*.

[22] Baolin Li, Tirthak Patel, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. 2022. Miso: Exploiting multi-instance gpu capability on multi-tenant gpu clusters. In *Proceedings of Symposium on Cloud Computing*.

[23] Junhan Liu, Zinuo Cai, Yumou Liu, Hao Li, Zongpu Zhang, Ruhui Ma, and Rajkumar Buyya. 2025. SMore: Enhancing GPU utilization in deep learning clusters by serverless-based co-location scheduling. *IEEE Transactions on Parallel and Distributed Systems*. 903–917.

[24] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative llm inference using phase splitting. In *Proceedings of Annual International Symposium on Computer Architecture*.

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

[26] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, fine-grained GPU sharing for ML applications. In *Proceedings of European Conference on Computer Systems*.

[27] Cheng Tan, Zhichao Li, Jian Zhang, Yu Cao, Sikai Qi, Zherui Liu, Yibo Zhu, and Chuanxiong Guo. 2021. Serving DNN models with multi-instance gpus: A case of the reconfigurable machine scheduling problem. *arXiv preprint arXiv:2109.11067* (2021).

[28] Qwen Team et al. 2024. Qwen2 technical report. *arXiv preprint arXiv:2407.10671* 2, 3 (2024).

[29] A. Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*. 5998–6008.

[30] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. 2023. Transparent GPU sharing in container clouds for deep learning workloads. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation*.

[31] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic scaling on GPU clusters for deep learning. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementations*.

[32] Jiazhi Yang, Shenyuan Gao, Yihang Qiu, Li Chen, Tianyu Li, Bo Dai, Kashyap Chitta, Penghao Wu, Jia Zeng, Ping Luo, et al. 2024. Generalized predictive model for autonomous driving. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*.

[33] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A native serverless system for low-latency, high-throughput inference. In *Proceedings of Architectural Support for Programming Languages and Operating Systems*.

[34] Bowen Zhang, Shuxin Li, and Zhuozhao Li. 2024. MIGER: Integrating multi-instance GPU and multi-process service for deep learning clusters. In *Proceedings of International Conference on Parallel Processing*.

[35] Renjun Zhang, Tianming Zhang, Zinuo Cai, Dongmei Li, and Ruhui Ma. 2025. MemoriaNova: Optimizing memory-aware model inference for edge computing. *ACM Transactions on Architecture and Code Optimization* 22, 1 (2025), 1–25.

[36] Jingyuan Zhao, Wenyi Zhao, Bo Deng, Zhenghong Wang, Feng Zhang, Wenxiang Zheng, Wanke Cao, Jinrui Nan, Yubo Lian, and Andrew F Burke. 2024. Autonomous driving system: A comprehensive survey. *Expert Systems with Applications* 242, Article 122836 (2024).

Received 11 September 2025; revised 12 April 2026; accepted 14 May 2026