Latency-Aware GPU Scheduling for DL Inference Tasks with Internal Resource Partitioning Strategy

Taewoo Kim, Changha Lee, Chan-Hyun Youn
School of Electrical Engineering
KAIST
Deajeon, Korea
{taewoo_kim, changha.lee, chyoun}@kaist.ac.kr

***Abstract*—With advancements in architecture and open-source foundation models extracting general knowledge, many developers are leveraging deep learning (DL) as application services. To coduct computation-intensive matrix operations, hardware resources with numerous parallel cores have also seen significant progress. Due to the high operational costs of such resources, efficient processing of DL tasks has become crucial for computing resource providers. Most studies on resource management for DL inference tasks focus on adjusting batch sizes to control the efficiency of a resource. Different from training tasks, however, inference processing uses relatively few internal resources, such as parallel cores and memory units. Consequently, a single inference task often fails to fully utilize the resource. Moreover, the latency prediction model, which is linearly proportional to batch size, still has limitations with small batch sizes and diverse architectures. In this paper, we introduce a novel GPU scheduler to improve processing efficiency by partitioning internal resources and executing multiple tasks simultaneously in a single GPU. Our latency model considers the internal kernel operation distribution, enabling precise prediction even in small batch sizes. By adjusting both batch size and usage ratio of internal resources the proposed scheduler can achieve the coexistence of multi-tenant DL tasks effectively. Experimental evaluations demonstrate that our approach achieves higher throughput and power efficiency compared to conventional scheduling mechanisms.**

***Index Terms*—deep learning, scheduling, GPU**

I. Introduction

The rapid advancements in the layered architecture [1]–[6] and the availability of pre-trained models with large-scale open datasets have significantly encouraged developers to leverage deep learning (DL) models for a variety of application services. Concurrently, hardware resources such as GPUs, FPGAs, and ASICs, which are designed to handle computation-intensive matrix operations, particularly those utilizing numerous parallel cores, have achieved high computational power. These single instruction multiple data (SIMD) cores can efficiently process large-scale matrix multiply-and-accumulate (MAC) operations in parallel. However, such hardware often incurs high operational costs, not only due to power consumption but also because of the substantial expenses of the hardware resources themselves. Meanwhile, the number of DL services is growing explosively, necessitating resources that can efficiently handle increasing user requests. Therefore, for service providers offering computing resources for DL, such as data centers, the primary objective is to efficiently accommodate multi-tenant DL services and manage their processing effectively.

Conventional approaches [7], [8] to resource management for DL inference tasks mainly focus on adjusting batch sizes to control the processing efficiency (i.e., utilization of parallel cores) of a resource. Increasing the batch size, which refers to the number of inputs processed at once, leads to higher amounts of matrix computation, thereby better exploiting the parallel cores. Although this is effective in managing DL services when scheduling, there are still limitations. The batch size, highly related to memory consumption during runtime, is constrained by off-chip memory capacity and latency service level objectives (SLOs), potentially resulting in restricting small batch size and underutilization of a resource. To address this issue, recent libraries often support the simultaneous execution of multiple tasks by partitioning internal resources such as parallel cores and memory unit. Recent studies [9]–[12] have actively explored scheduling mechanisms under concurrent task execution.

In this paper, we introduce a novel GPU scheduler designed to enhance processing efficiency with the partitioning of internal resources, enabling the allocation of multi-tenant DL tasks with various computational intensities. Our approach includes an advanced latency prediction model that considers the distribution of internal kernel operations by categorizing them into computation-intensive functions (MAC), and memory-intensive functions (element-wise operations). Then, the proposed scheduling engine determines the execution configuration, including *batch size* and *usage ratio*, which refers to the maximum portion of parallel cores participating in processing. This enables the simultaneous co-location of tasks with different architectures on the same GPU while maximizing processing efficiency.

In this paper, we discuss conventional scheduling mechanisms and observe the processing characteristics of different kernel functions used in DL. Then, we describe novel latency modeling and scheduling mechanisms under an environment with multi-tenant DL tasks. Finally, we demonstrate the effectiveness of the proposed scheduling approach through comprehensive experimental evaluations.

This work was supported by Samsung Electronics Co., Ltd (IO201210-07976-01).

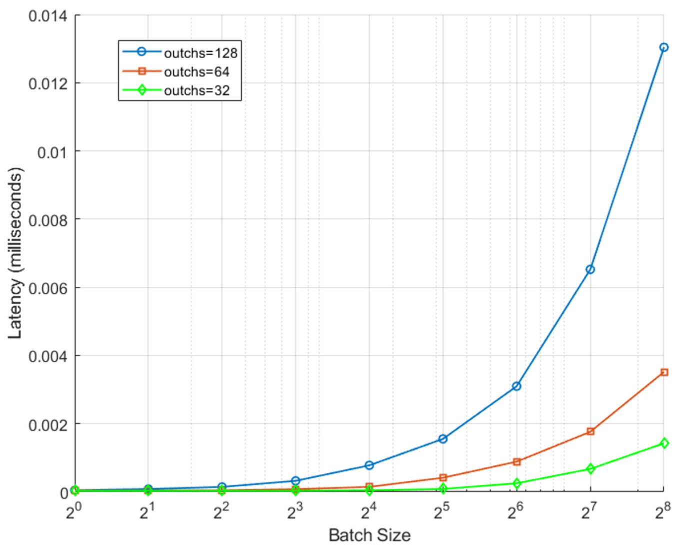

Fig. 1: Latency curve of *computation-intensive kernels* of convolution according to the output channels (outchs) and batch size of the input tensor

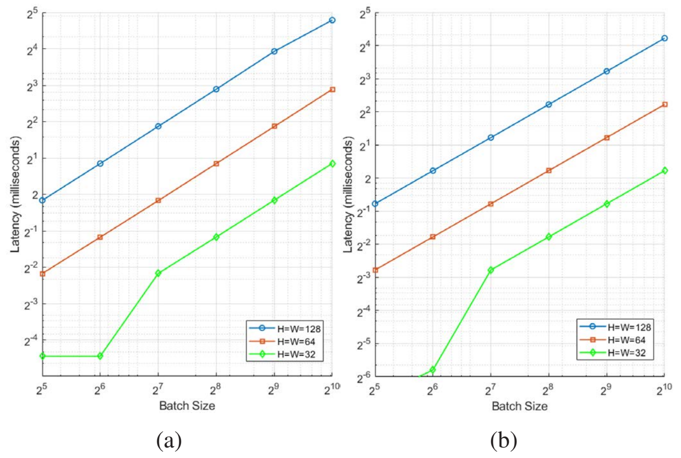

Fig. 2: Latency curve of *memory-intensive kernels* according to the width W, height H and batch size of the input tensor: (a) batch normalization and (b) ReLU.

II. Related Work and Problem Description

Traditional DL execution mechanisms involve dedicating all internal resources to a single task. The conventional studies regarding inference schedulers [7], [8], [10] have an allocation strategy with only a single task per resource with limited temporal multiplexing. On the other hand, recent libraries [13], [14] have been released, which allow multiple tasks to simultaneously employ internal resources via partitioning. In response to these advancements, recent studies, such as GSLICE [9] and gpu-let [12], have focused on how to split internal resources among multiple DL tasks. They either determine the usage ratio for partitions by adjusting in an online manner or predicting interference among allocated tasks.

However, there is still a lack of sophisticated modeling for latency prediction through detailed analysis of the internal kernel behavior, which is necessary for accurate resource allocation and multi-tenant DL architectures. It is composed of a combination of numerous types of layers. In this paper, we categorize them into *computation-intensive kernels*, which the computational load is dominant compared to the memory access of input and output data, and *memory-intensive kernels* in the opposite case. The former typically includes layers with matrix-matrix operations, while the latter usually contains element-wise operations or normalization functions. This categorization may achieve more precise latency modeling regardless of the model itself. We observed the tendency of these kernels to the batch size.

Fig. 1 shows that the latency of convolutional layers increases exponentially with the batch size. This behavior is due to the extensive matrix MAC operations. On the other hand, Fig. 2 demonstrates that memory-intensive operations such as batch normalization and ReLU. It implies that the latency is linearly proportional to batch size, driven by data transfer rates rather than arithmetic operations. We conduct the prediction modeling for completion time combined with an exponential and linear term, providing a foundational basis for efficient scheduling described in the following section.

III. Proposed latency-aware GPU Scheduler

A. Overall Architecture

In this section, we describe the overall architecture of the proposed latency-aware GPU scheduler, which is designed for multi-tenant DL services. Our scheduler can efficiently handle multiple GPUs by partitioning internal resources, allowing simultaneous execution of multiple DL tasks in a single resource. There are several mechanisms for splitting internal resources to allocate tasks, we assume the well-known method [13] of dividing parallel cores and leaving memory units to be shared over running tasks in this paper.

Fig. 3 represents an overall block diagram of the proposed scheduler. It consists of two function blocks: *the internal resource-ware latency predictor and resource scheduler*. The latency predictor estimates the completion time of processing a single batch of input requests while considering the kernel distribution of DL architecture and interference of shared resource units among concurrently running tasks, allowing the scheduler to anticipate a potential slowdown. The scheduler engine determines the execution configuration of partitions including a batch size and usage ratio of internal resources and the resource allocation strategy to maximize the processing efficiency of given DL tasks.

B. Internal Resource-Aware Latency Model

In this section, we delve into the prediction model of latency as *the completion time of processing a single batch of requests*, which is a core component of scheduling decisions. It considers not only the characteristics of kernels of layer operations in a target model but also accurate prediction when the task is executed in partial usage of parallel cores in GPU.

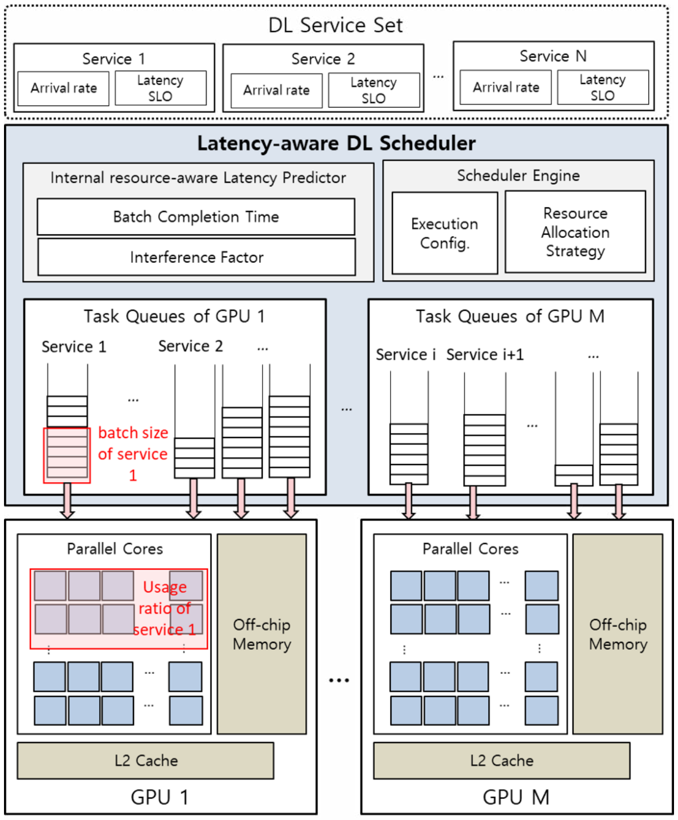

Fig. 3: Overall architecture of latency-aware GPU scheduler for multi-tenant DL inference services based on internal resource partitioning.

For a given DL task and a resource, we refer *base time*, t^{base} as the completion time of the model inference without any interference from other tasks on the execution resources, is given to Equation (1).

$$t^{base} = t^{queue} + t^{proc}, \tag{1}$$

where queue waiting time t^{queue} and processing time t^{proc}.

During inference, computationally intensive kernels $\mathcal{K}_{comp}$ such as convolution and linear operations, as well as memory-intensive kernels $\mathcal{K}_{mem}$ including element-wise and normalization operations, are iteratively processed within the resources. Then, we can represent the processing time t^{proc} as a function of the batch size $b \in \mathbb{N}$ and the usage ratio of internal resources (i.e. SIMD parallel cores), $0 \leq p \leq 1$ as follows.

$$\begin{aligned} & t^{proc}(b,p) \\ & = \left\{ \sum_{m \in \mathcal{K}_{comp}} t_m^{comp}(b) + \sum_{n \in \mathcal{K}_{mem}} t_n^{mem}(b) \right\} \cdot d(p), \end{aligned} \tag{2}$$

where $t_m^{comp}(b)$ and $t_n^{mem}(b)$ are completion time for m-th kernel in $\mathcal{K}_{comp}$ and n-th kernel in $\mathcal{K}_{mem}$, respectively. The degradation ratio $d(p)$ refers to the increase in completion time caused by using only a portion of the internal resources. When $p{=}1$, the degradation factor $d(p)$ is 1, and as p approaches 0, it increases. Note that due to the computational demands of the model, the increase in $d(p)$ is rarely until a certain point of p, but if it falls below that, the completion time increases sharply. Therefore, we can represent $d(p)$ with an exponential term of p with coefficients ξ and κ.

$$d(p) \approx e^{\xi(1-p)^{\kappa}}. \tag{3}$$

Meanwhile, it is impossible to accurately predict the individual completion times for tens of thousands of invoked kernels $(\mathcal{K}_{comp} \cup \mathcal{K}_{mem})$. Instead, we estimate the total completion time based on the characteristics of the layers of the given model and the batch size b, as shown in Equation (4).

$$\begin{aligned} & \sum_{m \in \mathcal{K}_{comp}} t_m^{comp}(b) + \sum_{n \in \mathcal{K}_{mem}} t_n^{mem}(b) \\ & \qquad \approx \left(\frac{b+\alpha}{e^{\beta \cdot b + \gamma}} + (\delta \cdot b + \omega) \right), \end{aligned} \tag{4}$$

where (α, β, γ) and (δ, ω) are regression coefficients for computation-intensive kernels $\mathcal{K}_{comp}$ and memory-intensive kernels $\mathcal{K}_{mem}$, respectively. Then, we can derive the prediction function for processing time $t^{proc}(b,p)$ using Equation (5).

$$t^{proc}(b,p) = \left(\frac{b+\alpha}{e^{\beta \cdot b + \gamma}} + (\delta \cdot b + \omega) \right) \cdot e^{\xi(1-p)^{\kappa}}. \tag{5}$$

Since we assume that processing time $t^{proc}(b,p)$ is almost deterministic for a given b and p, we can exploit the M/D/1 queueing model to estimate the queue waiting time t^{queue}. Note that the processing is conducted in a batching manner effective arrival rate becomes $\frac{\lambda}{b}$ in the system's perspective even though the arrival rate of a request is λ. Then we can derive $t^{queue}(\lambda, b, p)$ as Equation (6).

$$t^{queue}(\lambda, b, p) = \frac{\lambda \cdot (t^{proc}(b,p))^2}{2\,(b - \lambda \cdot t^{proc}(b,p))}. \tag{6}$$

Note that t^{base} implies the completion time when no other tasks are running on the resource. Let X as the set of tasks running simultaneously on a resource. Then, the completion time can be derived as the Equation (7).

$$\begin{aligned} & X = \{i | i\text{-th DL model running on a resource}\}, \\ & t^{con}(\lambda, b, p; X) {=} t^{queue}(\lambda, b, p) + t^{proc}(b,p) \cdot (1{+}I(X)), \end{aligned} \tag{7}$$

where interference factor $I(X) \in \mathbb{R}$ represents the overhead ratio due to contention of internal resources over running tasks. Inspired by gpu-let [12], we define it as the weighted sum of DRAM and L2 cache utilization of all tasks in an MPS [13]-based concurrent execution environment. Based on the latency modeling from above, we can derive the throughput of a task, as follows.

$$h(\lambda, b, p; X) = \frac{b}{t^{con}(\lambda, b, p; X)}. \tag{8}$$

Algorithm 1 Heuristics for Scheduling Algorithm on Multi-Tenant DL Tasks

```
Require: Set of inference services $\mathbf{S} = \{S_1, S_2, ..., S_N\}$,
    Number of GPUs $M$
Ensure: Execution configuration $\mathbf{C} = \{C_1, C_2, ..., C_N\}$, Resource allocation strategy $\mathbf{X} = \{X_1, X_2, ..., X_M\}$
 1: Initialize $\mathbf{C}$ with maximizing $b/t^{base}$
 2: Measure memory requirement of all services under $\mathbf{C}$
 3: Initialize $\mathbf{X}$ satisfying memory capacity
 4: repeat
 5:     for each GPU $j$ do
 6:         Calculate interference factor $I(X_j)$
 7:         for each task $i$ on GPU $j$ do
 8:             Calculate $T_i(C_i, X_j)$
 9:             Check $T_i(C_i, X_j) \leq L_i$ and $\frac{b_i}{T_i(C_i,X_j)} \geq \lambda_i$
10:             Adjust $b_i$ and $p_i$ to improve $E_i(C_i, X_j)$
11:         end for
12:     end for
13:     Adjust $\mathbf{X}$ satisfying memory capacity
14: until convergence or maximum iterations
15: return $\mathbf{C}$ and $\mathbf{X}$
```

C. Problem Statement for GPU Scheduling

In the previous section, we discuss the latency modeling of a DL task when multiple tasks are simultaneously running in GPU. This section describes the detailed scheduling problem of multi-tenant DL tasks.

At the time of the scheduling decision, there are N inference services $\mathbf{S} = \{S_1, S_2, ..., S_N\}$ and i-th service S_i has the following attributes:

$$S_i = <\lambda_i, L_i>, \tag{9}$$

where λ_i is an arrival rate of request and L_i is a latency SLO. In a computing environment handling these services, we assume M homogeneous GPUs as resources for simplicity. Our scheduler should decide *the execution configuration* $\mathbf{C} = \{C_1, C_2, ..., C_N\}$ over $\mathbf{S}$ and *the resource allocation strategy* $\mathbf{X} = \{X_1, X_2, ..., X_M\}$ over M GPUs. The execution configuration of i-th service, $C_i = <b_i, p_i>$, includes the batch size of b_i and usage ratio of parallel cores p_i. And, the resource allocation strategy of j-th GPU, $X_j = \{x_{j,1}, x_{j,2}, ..., x_{j,N}\}$, is represented by N-dimensional binary vector which $x_{j,k} = 1$ if k-th service is executed on j-th resource, otherwise 0. Then the completion time for a single batching of i-th service executed in j-th GPU under C_i and X_j is defined as:

$$T_i(C_i, X_j) := t^{con}(\lambda, b, p; X)|_{\lambda=\lambda_i,(b,p)=(b_i,p_i),X=X_j}. \tag{10}$$

Note that our scheduling objective is to enhance the processing efficiency of the overall service workers running on available GPUs by partitioning the internal parallel cores of GPU (i.e. usage ratio). Therefore, we define *processing efficiency*, $E_i(C_i, X_j)$, as the throughput (samples/sec) per unit of parallel cores for the i-th service under the execution configuration C_i and the concurrently running tasks in j-th GPU X_j.

$$E_i(C_i, X_j) := \frac{1}{p_i} \cdot \frac{b_i}{T_i(C_i, X_j)}. \tag{11}$$

We begin by formulating the optimization problem of the proposed scheduler. The objective function is to maximize overall processing efficiency while meeting the latency SLOs.

$$\arg\max_{\mathbf{C},\mathbf{X}} \sum_{(C_i,X_j)\in(\mathbf{C},\mathbf{X})} E_i(C_i, X_j) \tag{12}$$

$$\text{s.t.} \quad T_i(C_i, X_j) \leq L_i, \forall i \in \{1, 2, ..., N\}, \tag{13}$$

$$\frac{b_i}{T_i(C_i, X_j)} \geq \lambda_i, \forall i \in \{1, 2, ..., N\}, \tag{14}$$

$$\sum_{j=1}^{M} x_{j,k} = 1, \forall k \in \{1, 2, ..., N\}, \tag{15}$$

The constraints include a guarantee of latency SLO L_i, queue stable condition, and that a service worker can only be created in a single GPU space. This stems from the common characteristic that inference processing does not require high computational resources of more than one GPU different from training. Due to the complexity of the scheduling problem, we provide an overview of heuristic methods used in the scheduling process, as described in Algorithm 1.

It initializes the execution configuration (batch size and usage ratio of internal resources) by maximizing the throughput b/t^{base} when no other tasks are running in GPU. After measuring the memory requirement of tasks, it initializes the resource allocation strategy by allocating tasks into M GPUs. Then, The interference factor for each GPU is calculated by Equation (10), and execution configurations are iteratively adjusted to satisfy the latency SLO and queue stable condition depicted in Equation (13) and (14) and improve processing efficiency. This adjustment repeats until convergence or a

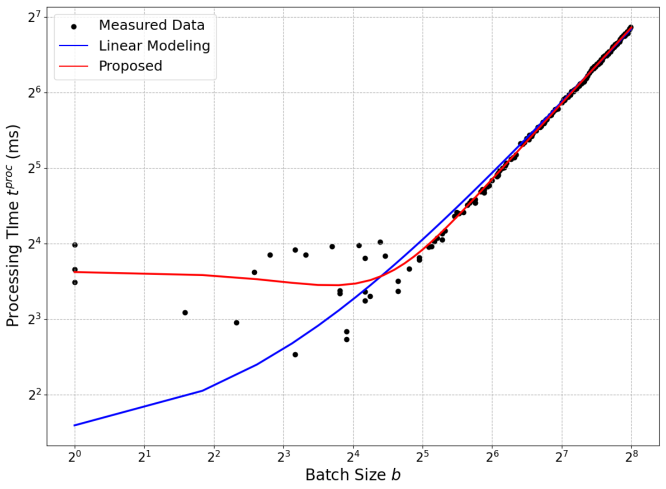

Fig. 4: Processing time prediction for varying batch sizes: measured data (black dot), the linear modeling (blue line) and the proposed modeling (red line).

TABLE I: Performance evaluation of task scaling with respect to scheduling mechanisms.

	Running Tasks	Power (W)	Throughput (reqs/sec)	Used GPUs
no partition	1	237.99	4328.71	1
	2	475.98	8657.42	2
	3	713.97	12986.13	3
	4	951.96	17314.85	4
w/ partition (base)	1	238.03	4325.86	1
	2	261.07	5030.7	
	3	253.59	5000.43	
	4	272.11	5086.19	
w/ partition (proposed)	1	238.31	4326.22	1
	2	277.15	5361.26	
	3	273.48	5278.5	
	4	290.26	5621.81	

maximum number of iterations is reached, ensuring efficient GPU scheduling.

IV. Performance Evaluation and Discussion

We conducted performance evaluations in our computing environment to observe the effectiveness of the proposed scheduler. Our experimental environment consists of servers equipped with four NVIDIA RTX 4080 GPUs, each featuring 16,384 parallel CUDA cores and 24GB of GDDR6 off-chip memory. They are interconnected via PCIe Gen3 x16 lanes. The host system is powered by an Intel 16-core Xeon Silver 4214R CPU running at 2.4GHz. We measured the performance of the proposed scheduling method (**w/ partition(proposed)**), which controls both batch size b and usage ratio p, against two baseline methods: one that allocates a single service per GPU and adjusts the batch size (**no partition**) [7], [8], and another that partitions the GPU without controlling the usage ratio p (**w/ partition(base)**)) [13]

A. Evaluation of Latency Prediction and Task Scaling

We first evaluated the processing time prediction model in Equation (4) and the resource efficiency when scaling the number of tasks with ResNet image classification model [1].

Fig. 4 shows the fitting of prediction models to processing times t^{proc} obtained from 200 randomly sampled batches. The conventional linear model used in [7], [8] assumes a linear relationship between batch size and processing time, resulting in a mean square error (MSE) of 6.57. On the other hand, the proposed non-linear model, which incorporates a non-linear relationship concerning computation-intensive and memory-intensive kernel functions, provides a better fit with a significantly lower MSE of 3.39. The non-linear model more accurately captures the processing times, especially for smaller batch sizes, and the sharp rise in processing times for larger batch sizes, illustrating its superiority in predicting processing times.

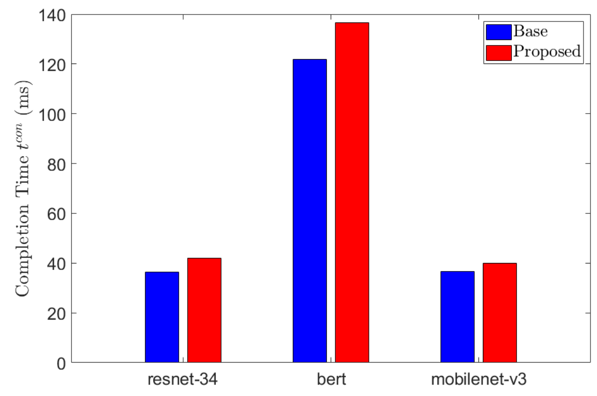

Fig. 5: Completion time with the proposed scheduling compared to the base partition method for individual tasks. Note that the latency SLO $L = [50, 200, 40]$, respectively.

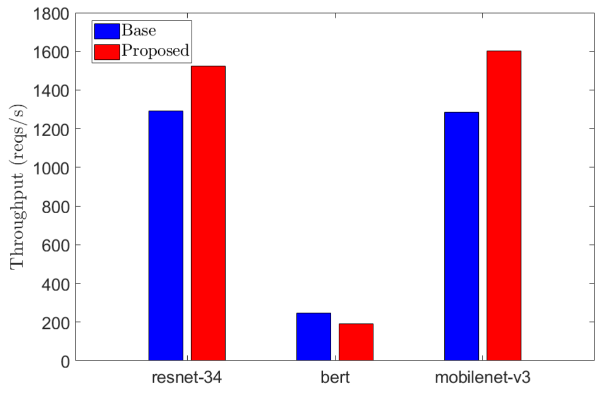

Fig. 6: Throughput (requests/sec) with the proposed scheduling compared to the base partition method for individual tasks.

Table I demonstrates the impact of task scaling on GPU requirements, power consumption, and throughput across different scheduling mechanisms. Without partitioning, as the number of tasks increases from 1 to 4, the required GPUs scale linearly from 1 to 4 GPUs, leading to a significant rise in power consumption from 237.99 W to 951.96 W. When partitioning is applied, both the base and proposed methods maintain a constant hardware requirement of only 1 GPU, significantly reducing power consumption compared to the no partition. Specifically, the proposed partitioning method shows a slight improvement in power efficiency and throughput compared to the base partitioning method, achieving a power consumption of 290.26 W and a throughput of 5621.81 reqs/sec for 4 concurrent tasks. The result implies the efficiency of partitioning methods in maintaining lower power consumption and GPU requirements while providing competitive throughput.

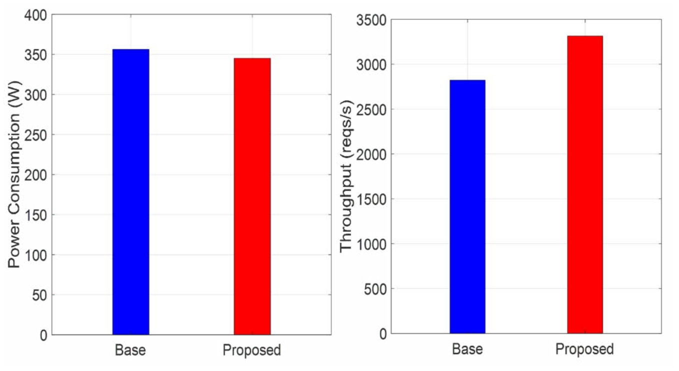

Fig. 7: Performance comparison between the proposed scheduling control (b, p) and the base partition method with fixed p: average power consumption (left) and total throughput for assigned tasks (right).

B. Evaluation of Proposed Scheduling under Multiple Tasks

Next, we aim to build multi-tenant DL services to demonstrate the effectiveness of the proposed scheduling. First, we set up different tasks (image classification, language processing) and prepared three model architectures (ResNet [1], MobileNet-v3 [15], BERT [16]). In the previous *no partition* approach, a single task could only be allocated to one GPU, which requires significant amounts of GPUs. However, by using the proposed partition method, all these tasks can be processed in a single GPU. In this setup, we set the arrival rates λ for each task to [200, 30, 600], and the latency SLOs L to [50, 200, 40] ms, respectively.

Fig. 5 and Fig. 6 show the completion time t^{con} of individual task depicted in Equation (7) and throughput of Equation (8). The base method, which multiple tasks can be executed without control of usage ratio p, fails to partition the internal resources adaptively, resulting in limitations on scheduling space of the batch size. Consequently, this leads to lower throughput over tasks except for BERT. In contrast, the proposed method adaptively partitions the internal resources by considering the computational and memory characteristics of each task by adjusting the usage ratio p. This adaptive allocation enables better handling of diverse workloads on the same GPU, achieving higher throughput while still meeting the latency SLO.

Fig. 7 demonstrates the average power consumption and total throughput over all tasks running in GPU. It implies that the proposed scheduler shows superior efficiency compared to the base partition method with a fixed $p = 1$. Specifically, the proposed method significantly reduces average power consumption, while increasing total throughput approximately increased by 15.9%. This result implies that the proposed method can appropriately allocate parallel cores based on the computational demands of the tasks and the arrival rate of an input service, thereby avoiding unnecessary contention among multi-tenant DL tasks.

V. Conclusion

In this paper, we proposed a novel GPU scheduling strategy designed to enhance the efficiency of DL inference tasks by partitioning internal resources to enable concurrent task execution. Experimental evaluations demonstrated that the proposed scheduling method improves processing efficiency, which reduces average power consumption while increasing total throughput compared to existing approaches. This is achieved through a more effective utilization of GPU resources, allowing multiple tasks to be processed on the same GPU without compromising performance. The proposed scheduling mechanism implies the potential of internal resource partitioning for a scenario with multi-tenant DL services.

References

[1] S. R. Kaiming He, Xiangyu Zhang and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.
[2] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.
[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.
[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*. Springer, 2015, pp. 234–241.
[5] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.
[6] L. Yao, C. Mao, and Y. Luo, "Graph convolutional networks for text classification," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, no. 01, 2019, pp. 7370–7377.
[7] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A gpu cluster engine for accelerating dnn-based video analysis," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 322–337.
[8] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A {Low-Latency} online prediction serving system," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 613–627.
[9] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "Gslice: controlled spatial sharing of gpus for a scalable inference platform," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 492–506.
[10] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving {DNNs} like clockwork: Performance predictability from the bottom up," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 443–462.
[11] P. Yu and M. Chowdhury, "Salus: Fine-grained gpu sharing primitives for deep learning applications," *arXiv preprint arXiv:1902.04610*, 2019.
[12] S. Choi, S. Lee, Y. Kim, J. Park, Y. Kwon, and J. Huh, "Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 199–216.
[13] NVIDIA multi-process service. https://docs.nvidia.com/deploy/mps.
[14] NVIDIA multi-instance gpu. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.
[15] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan *et al.*, "Searching for mobilenetv3," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2019, pp. 1314–1324.
[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.