2024 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS) | 979-8-3315-3130-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/MASCOTS64422.2024.10786549

Cloud Reamer: Enabling Inference Services in Training Clusters

Osama Khan
Sungkyunkwan University
Suwon, Republic of Korea
khan980@g.skku.edu

Gwanjong Park
Sungkyunkwan University
Suwon, Republic of Korea
jesj74@g.skku.edu

Junyeol Yu
Sungkyunkwan University
Suwon, Republic of Korea
junyeol.yu@skku.edu

Euiseong Seo
Sungkyunkwan University
Suwon, Republic of Korea
euiseong@skku.edu

***Abstract*—CPU cores in GPU servers are often underutilized during DNN training. Co-locating CPU-based inference tasks with DNN training offers an opportunity to utilize these idle CPU cycles. However, three technical challenges must be addressed: avoiding disruption to training workloads, meeting different performance requirements for online and offline inference, and swiftly adjusting inference configurations based on available resources. This paper proposes Cloud Reamer, a scheme to co-locate training and inference tasks on GPU servers, optimizing unused CPU cycles without disrupting training. Cloud Reamer prioritizes training tasks to minimize interference. For online inference, it allocates cores to ensure predictable performance, while for offline inference, it uses all available cores to maximize throughput. Cloud Reamer enhances online and offline inference performance by dynamically adjusting configurations based on surplus CPU resources. Evaluations show that Cloud Reamer improves inference throughput with minimal impact on training, maintaining training interference below 3.2%. It meets latency requirements for 46% more requests for online inference and achieves a 61x throughput increase for offline inference compared to conventional methods.**

***Index Terms*—co-location, deep neural networks, cloud computing, inference, training, interference**

I. Introduction

Training deep neural networks (DNNs) is mostly performed with GPU (graphic processing unit) servers equipped with multiple GPUs and tens of CPU cores. During the DNN training, GPUs execute the main computations, while CPU cores manage auxiliary tasks such as data preprocessing, GPU kernel dispatching, and gradient synchronization [1]. However, despite these auxiliary tasks, CPU utilization often remains low [1]–[3]. For example, in AISpeech's GPU cluster, the utilization of CPU cores is below 25% [1]. Similarly, within Alibaba's GPU clusters, 71% of GPU servers operate with an average CPU utilization rate of only 42% [3].

This underutilization of CPU cores in GPU servers stems from two reasons. First, securing sufficient PCIe bandwidth for each GPU often requires a specific ratio of CPU cores to GPUs [4], resulting in an excessive number of CPU cores for GPU-oriented workloads [3]. Second, users allocate more CPU cores than necessary [3]. This overestimation occurs due to a lack of precise knowledge about the CPU resources needed for their specific training jobs.

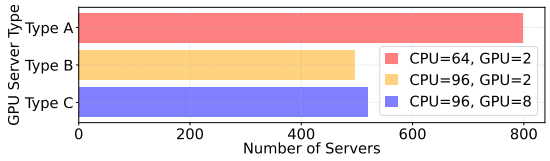
Unlike training, which requires GPUs, DNN inference can be efficiently executed on CPU cores. Leveraging this capability, co-locating a CPU-centric DNN inference workload with the training workload could be an intuitive approach to mitigate the underutilization of CPU cores [5]. However, fully exploiting the underutilized CPU cores requires overcoming three technical challenges that are currently unaddressed.

First, during training, CPU cores periodically handle tasks like preprocessing data, dispatching GPU kernels, and synchronizing gradients [1]. These tasks leave CPU cores idle from time to time, presenting an opportunity for inference workloads. However, simply co-locating inference workloads can disrupt the scheduling of training computations, extending the execution time of training tasks. Since training jobs are the primary focus of GPU servers, it is crucial to protect their performance while utilizing the otherwise wasted CPU cycles.

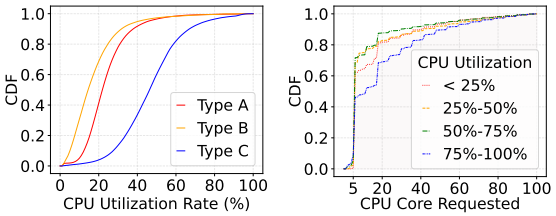
Second, inference workloads have different requirements [6]. Online inference demands real-time computation with strict latency constraints, so avoiding CPU contention with training jobs is more effective than fully utilizing spare CPU cycles. Offline inference aims to maximize throughput and can tolerate minor delays from training tasks, so using as many spare CPU cycles as possible is more effective. Therefore, handling inference workloads requires different co-location approaches based on their specific requirements.

Third, a DNN inference model can be deployed in various ways [7]. For example, we can adjust the number of independent instances, the CPU cores per instance, or the maximum batch size, creating numerous possible configurations. Additionally, the amount of available CPU cores frequently changes due to the sudden termination of training jobs [2]. Therefore, it is important to quickly fine-tune the parameters to maximize the throughput out of surplus CPU cores.

In this paper, we propose Cloud Reamer, a scheme designed to efficiently co-locate training and inference tasks on GPU servers, optimizing idle CPU cycles without compromising training performance. Cloud Reamer prioritizes training tasks to minimize scheduling disruptions. For online inference, it consolidates training tasks to a sufficient number of cores, allocating the remaining cores to inference to ensure consistent performance. For offline inference, Cloud Reamer utilizes all available cores, including those assigned to training tasks, to maximize throughput. Finally, Cloud Reamer quickly finds the inference job's configurations based on the runtime characteristics of a model and maximizes the efficiency of surplus CPU



(a) Server type distribution



(b) CPU Utilization (c) Requested CPU Cores

Fig. 1: Alibaba platform's GPU servers and their utilization.

resources for inference workloads.

II. Background and Motivation

A. Underutilization of CPU Cores in GPU Servers

Training is an iterative process that uses CPU cores for tasks such as preprocessing input data, dispatching kernels to GPU devices, and synchronizing gradients [1]. However, our analysis of publicly available GPU cluster logs [3], along with our local experiments, revealed that CPU cores remain underutilized in GPU servers despite these operations.

Specifically, CPU cores are underutilized in two ways. First, GPU servers' high CPU cores-to-GPU ratio often leads to underutilization. Fig. 1a depicts the distribution of nodes deployed in the Alibaba cluster [3], showcasing the number of CPU cores and GPUs associated with each server. The majority of servers belong to *Type A*, featuring 64 CPU cores and two GPUs, followed by *Type B* servers with 96 CPU cores and two GPUs and *Type C* servers with 96 CPU cores and 8 GPUs. Fig. 1b shows the CPU utilization per server type. *Type B* servers exhibit the lowest CPU utilization compared to *Type A* and *Type C*, while *Type C* shows the highest utilization rate among the three types. Consequently, GPU servers with higher CPU-to-GPU ratio had lower CPU core utilization.

Second, the lack of understanding of actual CPU core requirements by users further contributes to the underutilization of CPU cores. Fig. 1c shows the CDF of requested CPU cores among four different utilization groups. We observed that the majority of users allocated between 5 to 25 CPU cores, regardless of actual utilization. For example, 84% of requests for the group with less than 25% utilization were between 5 to 25 cores, with most requesting six cores. Similarly, 73% of requests for the group with more than 75% utilization were also between 5 to 25 cores, again with most requesting six cores. While high-utilization training jobs may justify a high number of CPU cores, the same allocation by low-utilization users indicates a lack of understanding of CPU requirements for training tasks. This consistent pattern of CPU core allocation across different utilization ranges suggests that users' allocations are often poor guesses of actual needs, further increasing idle CPU time in GPU servers.

TABLE I: DNNs Models Used for Training and Inference

Task	Model	Dataset	Preprocessing
Image Classification	VGG19 [8]	ImageNet	Normalization Image Flipping Image Resizing
	ResNet101 [9]		
	MobileNetV2 [10]		
Sequence Classification	Bert-Base [11]	CoLa	Sequence Padding Sequence Masking
	Bert-Large [11]		
Speech Recognition	DeepSpeech [12]	Common Voice	Mel Scale Audio Stretching Time-Freq. Masking

We further validated the underutilization of CPU cores in our local server. Table II describes the specification of this server. For our experiment, we selected six different models with specific settings outlined in Table I. To handle the preprocessing requirements, we adopted the reference implementation provided by MLPerf for training [13]. To verify our findings, we allocated six CPU cores and a single GPU for each model and trained every model one at a time.

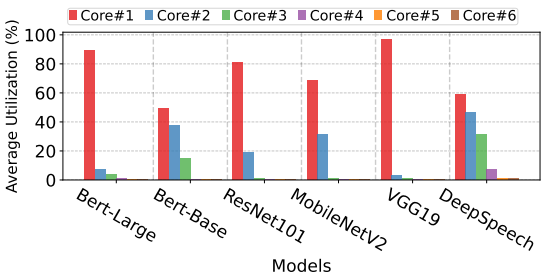


Fig. 2: Utilization of CPU cores during the training process.

The allocated CPU cores remained significantly underutilized during the training process. Fig. 2 shows the utilization of all the models when trained on a single GPU. Remarkably, none of the models fully utilized all the available CPU cores. The difference in utilization across the different models contributes to the fact that the larger the GPU time, the smaller the proportion of the preprocessing will be if the preprocessing time remains the same. In addition, text preprocessing is usually easier and simpler than image preprocessing, making the CPU side computation of the language model less CPU intensive [1]. However, regardless of these variations, all the models only utilized a fraction of the allocated CPU cores.

B. Challenges with Co-locating Inference Workloads

While DNN training workloads require GPUs for their intensive computations, DNN inference task is usually executed on CPU cores due to their large on-chip memory, support for vector operations, general-purpose computing capabilities, and mature software ecosystems that align with the portability and latency requirements of inference workloads [14]–[16]. Leveraging these capabilities, inference jobs are usually dispatched to CPU servers, while training jobs are dispatched to GPU servers [14]. However, given that the CPU cores of GPU servers are underutilized, co-locating a CPU-centric

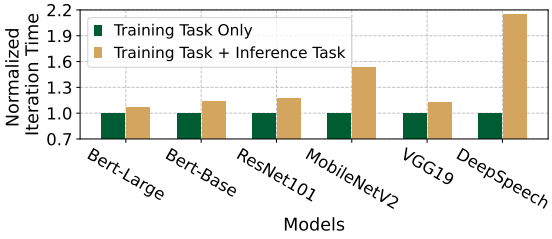


Fig. 3: Change in iteration time of training models when Bert-Base inference model is co-located on same CPU cores.

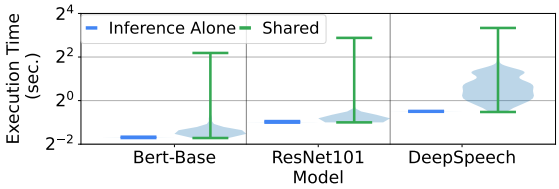


Fig. 4: Comparison of execution times when inference workloads are executed alone versus when they are co-located with training workloads.

DNN inference workload with the training workload could be an intuitive approach to mitigate this problem [5]. Furthermore, by co-locating inference jobs with the training job, we can reduce the cost associated with maintaining and using dedicated CPU servers. Nonetheless, fully exploiting these underutilized CPU cores requires overcoming three technical challenges that are currently unaddressed.

First, the training process intermittently uses the CPU cores for tasks such as preprocessing input data, gradient synchronization, and kernel dispatching, leaving the CPU cores idle at times. The inference workload can utilize these idle periods. However, when an inference workload is co-located on the same CPU cores as the training workloads, the contention for CPU time can prolong the execution time of the training jobs.

To demonstrate this, we co-located a Bert-Base inference model on the same CPU cores with the training workloads presented in Table I. As shown in Fig. 3, this co-location led to longer training times despite the CPU cores' significant underutilization during training. For example, MobileNetV2 and DeepSpeech experienced 54% and 114% increases from their original execution time, respectively. Furthermore, the extent of interference on the training workload depends on the CPU-side computational demands of the training process. For example, DeepSpeech requires intensive audio preprocessing. Any delay in these CPU-side computations can significantly extend the overall execution time. In contrast, MobileNetV2 requires less preprocessing, but its smaller model size allows it to be processed quickly. As a result, the CPU must dispatch kernels and perform the gradient updates without any delay.

Second, applying a uniform strategy to utilize surplus CPU cores for inference workloads can compromise the potential benefits. Specifically, inference workloads are of two types [6]. Online inference workloads, such as online translation, require completing the inference process within a specific time interval. These workloads experience fluctuating request rates over time, yet all responses must adhere to the predefined time constraints. Conversely, offline inference workloads aim to maximize throughput without timing constraints. An example is facial recognition in a photo album. In this case, the system is free to process the data in any order, and therefore, the aim is to maximize the throughput as much as possible.

Co-locating online inference workloads with training tasks on the same CPU cores can lead to unpredictable latency, making this approach unsuitable due to their strict timing requirements. To show this, we conducted an experiment using CPU-centric DNN inference models, Bert-Base, ResNet101, and DeepSpeech and executed them on the same CPU cores allocated for training. As shown in Fig. 4, when executed alone, the standard deviations in the measurement of Bert-Base, ResNet101 and DeepSpeech were only 0.002, 0.0012, and 0.001 seconds, respectively. However, when co-located with the training job, the standard deviations were approximately 0.342, 0.188, and 0.811 seconds, respectively. This is because the exact time for the inference job to get a CPU core is undetermined due to contentions made by the training job. This variability in execution time, which results in inconsistent and unpredictable outcomes, cannot be tolerated by online inference workloads that operate under tight time constraints. Nonetheless, such a co-location strategy offers the potential to increase throughput for offline inference workloads that lack such timing constraints, thereby highlighting the necessity for tailored strategies based on their specific requirements.

Third, CPU-based inference models have numerous configurable parameters. These parameters include the number of independent instances, the cores per instance, and the maximum batch size allowed per instance. When deploying an inference workload, we need to maximize the throughput out of surplus CPU cores. However, profiling all potential configurations is impractical due to their vast search space. Consider a situation with 32 idle CPU cores and a maximum batch size of 32. We would need to evaluate 267,168 possible configurations derived from 8349 possible integer partitions of 32 CPU cores (possible instances) multiplied by 32 batch sizes, making profiling an unfeasible approach.

In addition to this, the surplus CPU resources of a GPU server may abruptly change. While the completion time of a training job can be known in advance, these jobs can be stopped at any time. For example, in large training clusters, about 30.7% of tasks are ended earlier, with the majority ending in less than 10 minutes [2]. If an ongoing training job terminates and a new training job with different CPU core requirements arrives, we must again find the configuration that will maximize the throughput for the changed CPU cores. Therefore, it is essential to be able to quickly customize these parameters for the given available CPU cores.

III. Our Approach

We designed Cloud Reamer, a set of strategies to co-locate training and inference jobs on GPU servers, utilizing wasted

CPU cores. Cloud Reamer aims to minimize interference with training workloads, provide different co-location strategies based on the specific requirements of inference workloads, and maximize throughput from surplus CPU cores.

A. Minimizing Interference to Training Jobs

When inference jobs are co-located with training jobs, they can be scheduled on the same CPU cores, leading to contention for CPU time. This increases the duration of the training process and causes inconsistent times for online inference jobs. To avoid this, Cloud Reamer consolidates training processes onto specific cores, reserving others for inference jobs.

Cloud Reamer maintains a table of CPU cores' allocation status. When no training jobs are running, all CPU cores are available. Upon receiving a training job, Cloud Reamer identifies the number of cores to be allocated for the job by examining the number of cores assigned to the container, sequentially selects cores from the unallocated list, and assigns them to the container for the training workload. These cores are bound to the training job using the cgroup's cpuset feature provided by Linux. The core status table is then updated with the training job's ID and the allocated CPU cores.

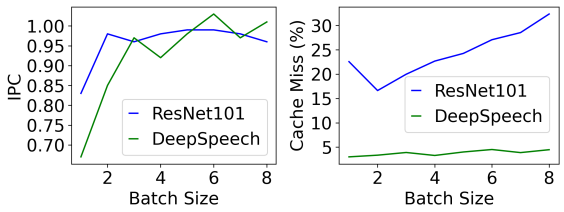
This consolidation minimizes interference with training workloads and leaves idle cores available for inference jobs. By co-locating an online inference job on these idle cores, we can ensure predictable execution times. In addition, training processes use CPU cores for tasks such as preprocessing input data, gradient synchronization, and kernel dispatching, leaving the CPU cores idle at times. Therefore, after consolidating the training processes, Cloud Reamer co-locates offline inference jobs with the training process and utilizes the available CPU cycles within the consolidated CPU cores.

To mitigate CPU time interference within the consolidated CPU cores and allow the offline inference workload to utilize the sporadic CPU cores, Cloud Reamer prioritizes training jobs to shared CPU cores by running the training processes in a real-time (RT) scheduling class. The higher priority of the RT scheduling class allows the training processes to preempt the inference jobs and utilize the CPU cores as needed. This is done using the *chrt* command in Linux. Furthermore, Cloud Reamer opts for a round-robin sequence of execution to prevent any single process from monopolizing the system. Note that ensuring fair process execution time within the RT scheduling class requires a more sophisticated approach, which remains beyond the scope of this paper.

B. Maximizing Performance for Inference Jobs

After separating the idle CPU cores from the ones that are specified for the training workload, Cloud Reamer aims to maximize the throughput for the co-located inference jobs. However, maximizing throughput out of the completely idle CPU cores requires evaluating various configurations, which can be time-consuming. To address this, Cloud Reamer employs a quick search method.

We observe that increasing batch size is a common approach to boost throughput, but its effectiveness varies across models.



(a) Instruction Per Cycle (b) Cache Miss Percentage

Fig. 5: IPC and cache miss percentage for ResNet101 and DeepSpeech under different batch sizes.

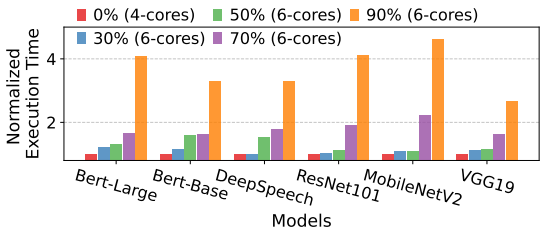


Fig. 6: Execution times when co-located with a single varying-utilization task versus using only four exclusive cores.

To investigate this, we looked at two different characteristics of a model: the CPU core utilization and the data reusability. We measured the IPC for CPU core utilization and cache miss rates for data reusability. We measured these two metrics for ResNet101 and DeepSpeech as we increased the batch sizes. As shown in Fig. 5, DeepSpeech maintained a low cache miss rate of around 5%, indicating better data reusability and efficient cache utilization. As batch size increased, its IPC also improved, showing better CPU core utilization. In contrast, ResNet101 started with a higher cache miss rate and IPC value. Initially, increasing the batch size improved both metrics, but further increases led to a higher cache miss rate while IPC remained unchanged. This suggests that for ResNet101, larger batch sizes do not enhance data reuse or CPU utilization. We observed similar behavior for other models as well.

Specifically, for models with a stable cache miss rate and increasing IPC as batch size increases, creating a single large instance and allocating all cores to it can enhance throughput. Conversely, for models that show increasing cache miss rates and declining IPC with larger batch sizes, creating multiple instances can enhance performance. Based on these observations, instead of evaluating all possible configurations, Cloud Reamer uses the model's runtime characteristics to decide whether to start the search with small instances or a single large instance.

The number of cores per instance and the batch size to start the quick search depends on the type of inference workload. For online inference workloads, it is crucial that each instance meets latency constraints. Therefore, if a model's runtime characteristics show that it can benefit from a large number of instances, Cloud Reamer first aims to find the minimum number of CPU cores per instance to ensure that every allocated instance will perform within the latency constraint.

To do this, Cloud Reamer starts by creating a single instance on one core, gradually increasing the number of cores until the latency constraint is satisfied. Once the minimum number of cores required per instance is determined, Cloud Reamer starts the quick search method to maximize throughput. Meanwhile, for offline inference workloads, Cloud Reamer directly starts searching for the maximum throughput. Specifically, after finding the cache miss rate and IPC values, the search starts from a large number of instances, each with a single CPU core or a single instance with the maximum batch size.

Furthermore, Cloud Reamer executes offline inference tasks on the consolidated CPU cores for the training workloads, deploying either a single instance or multiple instances on these cores. However, co-locating a single instance and allocating all the CPU cores can increase the execution time for the inference job due to the interplay between the threads, impacting the overall performance of the inference job. To show this, we allocated six CPU cores for the inference workload and executed an additional CPU task on one of the cores, simulating the CPU-side computation of a training process. We employed this CPU task to modulate the available time within that particular CPU core. For example, we designed this task to occupy only 30% of the CPU core's time, remaining idle for the other 70% of the time. This resulted in the inference task fully utilizing 100% of the other five cores while only utilizing 70% of the first core. We varied the duration of time available for the inference tasks on this single CPU core. As a base case, we compared the results when inference workloads are executed solely on four CPU cores.

Fig. 6 illustrates the adjusted execution time of inference workloads for various models when co-located with a single varying-utilization task versus using only four exclusive cores. Despite modifying the available CPU time for one core and allowing it to utilize five CPU cores fully, we observed that the overall performance of the inference task worsens compared to utilizing four CPU cores exclusively. This is because even if all threads finish execution except for one, the final execution time of an operation is determined by the slowest thread, which could be held up by a training process on a core. Therefore, instead of creating a single instance with large threads to utilize the allocated CPU cores, Cloud Reamer creates multiple instances with single threads for each instance, removing the bottleneck from the slow execution of the straggler thread. Note that for allocating an instance to an online inference job, the number of threads is same as the number of CPU cores.

C. Integration into Existing GPU Clusters

As depicted in Fig. 7, Cloud Reamer can be integrated into clusters, containing both CPU and GPU servers through two components: Cloud Reamer Agent (CR-Agent) and Cloud Reamer Extension (CR-Extension).

CR-Agent is responsible for minimizing interference and maximizing throughput for inference workloads. Specifically, CR-Agent has three main responsibilities: first, upon receiving a training job in a GPU server, it allocates the job to specific CPU cores and claims the surplus CPU cores. Second, it maximizes the performance of these surplus CPU cores. Finally, it ensures there is no interference with the training workload. On the other hand, CR-Extension is responsible for reassigning an inference job from CPU server to CPU cores of GPU servers. CR-Extension can be a policy embedded in the current cluster schedulers, e.g., as a plugin in Kubernetes scheduler [17].

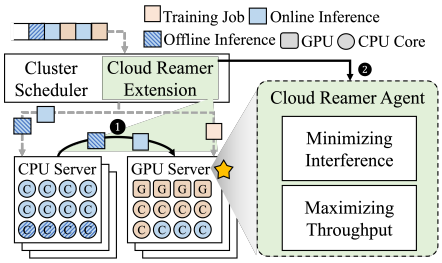


Fig. 7: Integration Cloud Reamer into existing GPU clusters.

TABLE II: Experimental Server Configurations

CPU	2 x Intel Xeon Gold 5128 @ 2.3 GHz 16 cores each, 2 logical threads per core
Memory	4 x 32 GB DDR4 @ 2400 MHz
GPU	4 x Nvidia Titan RTX 24 GB DDR6 per each
OS	Ubuntu 22.04

Initially, inference jobs are assigned to CPU servers and training jobs to GPU servers by the cluster scheduler. When all GPUs are occupied or there are no pending training jobs, CR-Extension identifies the online inference job with the lowest demand in CPU servers and transfers it to CR-Agent, which assesses whether this inference job can be executed on the GPU server by evaluating the IPC and cache miss rate of the model, seeking an optimized configuration by adjusting the batch size and number of instances. If the surplus CPU cores can provide sufficient throughput for the online inference job, CR-Agent will undertake the job and CR-Extension will remove it from the CPU server.

While the inference job is running on the GPU server, CR-Agent monitors the rate of incoming online inference requests. If this rate surpasses the rate for which the inference instance is initially configured, CR-Agent alerts CR-Extension, which then moves the job back to the CPU server. Conversely, if there are no suitable online inference jobs that can utilize the spare CPU cores, CR-Extension selects an offline inference job with less stringent latency requirements and forwards it to CR-Agent, which then processes the job. CR-Agent first maximizes the throughput for these jobs out of completely idle CPU cores and then also dispatches multiple instances on the CPU cores consolidated for the training jobs. To avoid interference with the training jobs, CR-Agent changes the scheduling class of the training jobs to the RT class.

IV. Evaluation

A. Evaluation Environment

We evaluated our approach using the GPU server described in Table II. For training, we used PyTorch version 2.0.0

with Nvidia CUDA support enabled, and for inference, we used PyTorch version 1.13.1, optimized for Intel CPUs. We assumed inference jobs were ready to be executed on a GPU server, and the relocation process was carried out manually.

We first evaluated the predictability when meeting the latency constraint of online inference jobs by comparing Cloud Reamer with three other co-location strategies. The first setup involved executing inference tasks solely on 16 CPU cores, serving as the base case for predictability (Inference Alone). In the second configuration, inference tasks were co-located with the training job on 32 CPU cores without being bound to specific cores, allowing threads to move freely but potentially contending for CPU time, referred to as *No Consolidation*. The final arrangement involved explicitly binding both training and inference jobs to 16 shared CPU cores, representing the worst-case situation, referred to as *Shared*.

We also evaluated Cloud Reamer's effectiveness in maximizing throughput under latency constraints for online inference workloads, comparing its performance to using the maximum batch size. This evaluation was conducted across 4, 8, and 16 CPU cores, with latency constraints ranging from 200 milliseconds to 1 second. The search time to find the maximum throughput under all the latency constraints on 4, 8 and 16 CPU cores took approximately 13 minutes for DeepSpeech, 12 minutes for Bert-Base, and 10 minutes for ResNet101. During both the predictability and throughput evaluations for online inference workloads, we executed DeepSpeech training workloads on 16 CPU cores and allocated all four GPUs. The batch size for this training workload is set to 32.

Second, we measured the throughput achievable by offline inference workloads using sporadic surplus CPU cycles. We compared Cloud Reamer with three methods: by using the default scheduling policy, allocating a single large instance on the same CPU cores as the training workload and using gang scheduling. Gang scheduling enables exclusive execution of a given workload at any given time, allowing for division of CPU time between two applications. In our situation, this approach lets the training process utilize the cores, and for brief intervals, it preempts them to run only the inference processes. To implement gang scheduling, we measured the average CPU core utilization rate over one minute and used it to determine the inference workload's execution time. Given the heavy CPU-sided computation of DeepSpeech, we selected this model for evaluation. Specifically, we allocated 16 CPU cores and 4 GPUs for the DeepSpeech model. We allowed the co-located inference jobs to be executed only for 0.2 seconds while the training jobs were executed for 0.8 seconds. The batch size for offline inference workloads was set to one. This was implemented using the freezer functionality from cgroup. Finally, we measured the contention to this DeepSpeech training model when offline inference workloads were executed on the same set of CPU cores.

B. Performance Analysis

Fig. 8 shows the variance in execution time under different co-location strategies for Bert-Base, ResNet101, and DeepSpeech models. Co-locating on the same set of CPU cores (Shared) resulted in the highest variance, with standard deviations of 0.742, 1.404, and 1.367 respectively for Bert-Base, ResNet101, and DeepSpeech. Allowing processes to move freely among the CPU cores (No Consolidation) reduced interference but still resulted in significant variance, with standard deviations of 0.011 for Bert-Base, 0.019 for ResNet101, and 0.056 for DeepSpeech. Cloud Reamer, which binds each job's processes to specific cores, significantly reduced variance, resulting in standard deviations of 0.006 for Bert-Base, 0.005 for ResNet101, and 0.008 for DeepSpeech. This is due to the job consolidation which not only allows the training workloads to execute without any interference but also provides a stable environment for inference workloads, demonstrating Cloud Reamer's effectiveness in providing predictable performance.

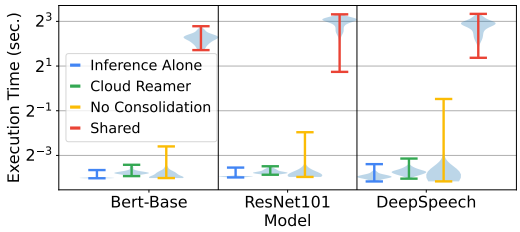
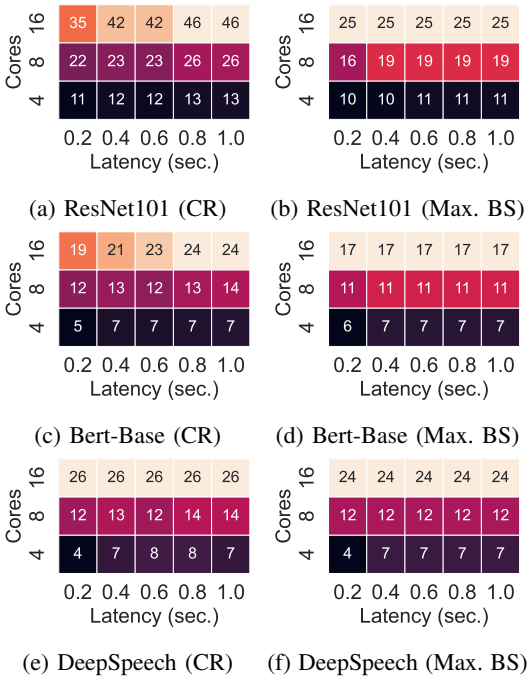


Fig. 8: Comparison of execution times when inference workloads are executed alone with different co-location method.



(a) ResNet101 (CR) (b) ResNet101 (Max. BS)

(c) Bert-Base (CR) (d) Bert-Base (Max. BS)

(e) DeepSpeech (CR) (f) DeepSpeech (Max. BS)

Fig. 9: Maximum throughput under varying latency constraints and CPU cores using Cloud Reamer and maximum batch size.

Fig. 9 demonstrates that Cloud Reamer consistently outperforms the maximum batch size approach in achieving higher throughput under various latency constraints. Specifically, using Cloud Reamer led to an adjustable throughput based on

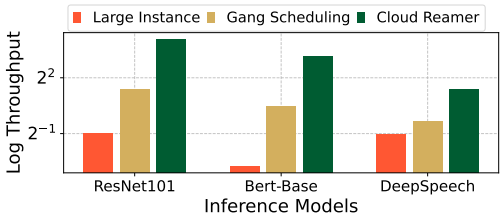


Fig. 10: Comparison of throughput when CPU cores are shared with the DeepSpeech training workload.

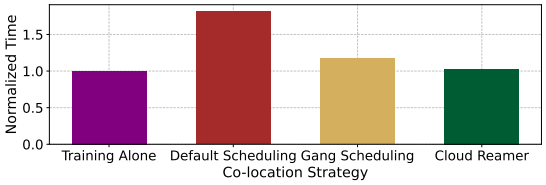


Fig. 11: Variation in DeepSpeech model training iterations when inference workloads shared the same CPU cores.

latency constraints; for instance, with ResNet101, throughput varied from 35 to 46 as latency constraints were altered. In contrast, the throughput remained constant when employing the maximum batch size strategy. This illustrates that solely increasing the batch size can only enhance throughput to a certain limit. Once the optimal throughput is achieved with a specific batch size, further increases do not impact throughput. Thus, for these models, opting for multiple instances over larger batch sizes proves advantageous. However, only a modest improvement was noted with the DeepSpeech model using Cloud Reamer, attributed to the model's preference for large batch sizes to maximize throughput. On average, Cloud Reamer delivered throughput improvements of 45%, 20%, and 5% over the strategy of always using the maximum batch size.

Fig. 10 illustrates the throughput when three offline inference models were run concurrently with the DeepSpeech training workload on the same CPU cores. Cloud Reamer outperformed both gang scheduling and deployment on a single large instance due to its capability to deploy multiple instances independently. This design allows other instances to continue processing if one is hindered by the training workload, thus optimizing CPU core utilization.

Fig. 11 illustrates the variations in training execution time for the co-located DeepSpeech model under different co-location strategies. When using the default scheduling policy, the execution time increased by approximately 81%. Gang scheduling reduced this increase to 17%, and with Cloud Reamer, the interference to the ongoing DeepSpeech training job was minimized to just 3.2%. These results indicate that Cloud Reamer effectively reduces interference by prioritizing training workloads on CPU cores while utilizing available CPU cycles simultaneously.

V. Related Work

The practice of co-locating CPU workloads with GPU workloads to enhance CPU utilization has been explored in several studies [1], [5], [18].

Jiadong et al. [18] developed a method that reallocates leftover CPU resources from GPU-based HPC tasks to CPU-based tasks. They statically assigned CPU cores based on the core usage of GPU tasks observed during a monitoring period. CODA [1] is a scheduling scheme for GPU clusters that optimizes resource utilization by identifying the minimum number of cores needed to maintain training performance and using surplus cores for CPU-based inference jobs. SODA [5], an extension of CODA, enhances efficiency by deploying inference workloads on GPUs during high load and on CPUs during low load. These approaches allocate the minimum number of cores necessary to avoid degrading the performance of GPU workloads and use the remaining cores for CPU tasks. This is based on the assumption that the optimal number of CPU cores for a given GPU workload is fixed. However, a training workload has highly variable CPU demands depending on its phase, such as preprocessing and GPU kernel launches. Consequently, the core allocation proposed by these approaches still leads to significant CPU cycle wastage due to these large fluctuations in CPU core requirements. Additionally, SODA deployed CPU inference workloads with the maximum batch size possible, assuming it would yield the highest throughput. However, our research found that optimizing CPU inference workloads, by increasing the number of instances even with smaller batch sizes depending on the target model, can lead to better performance with the same number of CPU cores.

PARTIES co-locates multiple latency-critical services on a cloud node, dynamically reallocating resources, such as memory bandwidth and last-level cache, to meet their quality of service (QoS) [19]. CLITE [20] enhances PARTIES with a Bayesian Optimization-based multi-resource partitioning technique that simultaneously satisfies the QoS requirements of multiple co-located latency-critical jobs while maximizing the performance of co-located batch jobs. Ting et al. [21] used linear regression to model and predict performance interference between batch and latency-critical workloads, and also proposed a scheduler that deploys batch jobs only when the predicted interference with online jobs is within a predefined threshold. Kelp [22] addresses the performance degradation of accelerated machine learning (ML) tasks caused by memory bandwidth contention. It isolates high-priority ML tasks from resource interference, ensuring that the performance and efficiency gains of ML accelerators, such as GPUs, are preserved.

Our research prevents performance degradation of training workloads caused by scheduling delays at the OS level. However, it does not address the performance degradation due to increased hardware resource contention from co-location. According to our observations, the impact of hardware resource contention is minimal compared to scheduling interference. Nonetheless, our approach could provide more comprehensive performance guarantees for training workloads when combined with the introduced resource partitioning schemes.

NeoCPU [15] is a comprehensive and systematic optimization approach for CNN model inference on CPUs, using template-based operations and joint operation- and graph-

level optimizations without relying on third-party libraries. NIOT [16] is a general framework for optimizing inference of Transformer-like models on CPUs, leveraging modern CPU features such as SIMD and cache hierarchy. AsyMo [23] enhances performance scalability and energy efficiency for inference on asymmetric mobile CPUs through execution determinism, cost-model-directed partitioning, asymmetry-aware task scheduling, and energy-efficient frequency determination.

VI. Conclusion

Training GPU clusters exhibit low CPU utilization due to training workloads' high dependency on GPUs and inherently high CPU cores-to-GPU devices ratio. In such cases, deploying CPU-based inference workloads on GPU servers assigned for training jobs emerges as a natural but logical solution. However, integrating this approach into actual production systems requires ensuring predictable latency and maximum throughput for inference workloads without compromising training workloads' performance.

This paper proposed Cloud Reamer, a set of strategies designed to tackle the previously mentioned technical challenges by ensuring that both online and offline inference tasks efficiently utilize surplus CPU cycles without compromising the performance of training workloads. Our evaluation showed that Cloud Reamer minimizes interference with training processes to under 3.2% by strategically co-locating inference workloads. When compared to conventional methods for running both types of workloads in a single server, Cloud Reamer could accommodate 46% more online inference requests while satisfying their latency requirements. For offline inference tasks, Cloud Reamer realized a throughput enhancement of up to 61 times compared to the default deployment policy.

Acknowledgment

This research was partially supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2024-RS-2023-00258649) supervised by the IITP (Institute for Information & Communications Technology Planning & Evaluation), and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2023-00321688).

References

[1] H. Zhao, W. Cui, Q. Chen, J. Leng, K. Yu, D. Zeng, C. Li, and M. Guo, "CODA: Improving resource utilization by slimming and co-locating DNN and CPU jobs," in *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 2020, pp. 853–863.

[2] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant GPU clusters for DNN training workloads." in *Proceedings of the USENIX Annual Technical Conference*, 2019, pp. 947–960.

[3] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2022, pp. 945–960.

[4] *NVIDIA-Certified Systems Configuration Guide for PCIe Servers, DG-10105-001 _v05*, NVIDIA, 2023.

[5] H. Zhao, W. Cui, Q. Chen, J. Leng, D. Zeng, and M. Guo, "Improving cluster utilization through adaptive resource management for DNN and CPU jobs co-location," *IEEE Transactions on Computers*, vol. 72, no. 12, pp. 3458–3472, 2023.

[6] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou *et al.*, "MLPerf inference benchmark," in *Proceedings of the ACM/IEEE International Symposium on Computer Architecture*, 2020, pp. 446–459.

[7] OpenVINO, "Optimize Inference." [Online]. Available: https://docs.openvino.ai/2024/openvino-workflow/running-inference/optimize-inference.html

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

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[10] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.

[11] 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.

[12] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates *et al.*, "Deep Speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[13] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A GPU cluster manager for distributed deep learning," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*, 2019, pp. 485–500.

[14] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied machine learning at Facebook: A datacenter infrastructure perspective," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2018, pp. 620–629.

[15] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on CPUs," in *Proceedings of the USENIX Annual Technical Conference*, 2019, pp. 1025–1040.

[16] Z. Zhang, Y. Chen, B. He, and Z. Zhang, "NIOT: A novel inference optimization of transformers on modern CPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 6, pp. 1982–1995, 2023.

[17] Kubernetes, "Scheduling Frameworks." [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

[18] J. Wu and B. Hong, "Collocating CPU-only jobs with GPU-assisted jobs on GPU-assisted HPC," in *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, 2013, pp. 418–425.

[19] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 107–120.

[20] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2020, pp. 193–206.

[21] T. Zhang, D. Ou, Z. Ge, C. Jiang, C. Cérin, and L. Yan, "Interference-aware workload scheduling in co-located data centers," in *Proceedings of the IFIP International Conference on Network and Parallel Computing*, 2022, pp. 69–82.

[22] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: QoS for accelerated machine learning systems," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*, 2019, pp. 172–184.

[23] M. Wang, S. Ding, T. Cao, Y. Liu, and F. Xu, "AsyMo: scalable and efficient deep-learning inference on asymmetric mobile CPUs," in *Proceedings of the Annual International Conference on Mobile Computing and Networking*, 2021, pp. 215–228.