KACE: Kernel-Aware Colocation for Efficient GPU Spatial Sharing

Bing-Shiun Han, Tathagata Paul, Zhenhua Liu, Anshul Gandhi
Stony Brook University
Stony Brook, New York, USA
{bing-shiun.han,tathagata.paul,zhenhua.liu,anshul.gandhi}@stonybrook.edu

Abstract

GPU spatial sharing among jobs is an effective approach to increase resource utilization and reduce the monetary and environmental costs of running deep learning workloads. While hardware support for GPU spatial sharing already exists, accurately predicting GPU interference between colocated workloads remains a concern. This makes it challenging to improve GPU utilization by sharing the GPU between workloads without severely impacting their performance. Existing approaches to identify and mitigate GPU interference often require extensive profiling and/or hardware modifications, making them difficult to deploy in practice.

This paper presents KACE, a lightweight, prediction-based approach to effectively colocate workloads on a given GPU. KACE adequately predicts colocation interference via exclusive kernel metrics using limited training data and minimal training time, eliminating the need for extensive online profiling of each new workload colocation. Experimental results using various training and inference workloads show that KACE outperforms existing rule-based and prediction-based policies by 16% and 11%, on average, respectively, and is within 10% of the performance achieved by an offline-optimal oracle policy.

CCS Concepts

• **Computer systems organization** → **Cloud computing**.

Keywords

Cloud Computing, Systems for ML, GPU Sharing

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11
https://doi.org/10.1145/3698038.3698555

ACM Reference Format:
Bing-Shiun Han, Tathagata Paul, Zhenhua Liu, Anshul Gandhi. 2024. KACE: Kernel-Aware Colocation for Efficient GPU Spatial Sharing. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3698038.3698555

1 Introduction

Applications of Deep Learning (DL) models, such as image processing and speech recognition, have exponentially grown in recent years [19], resulting in useful services and products [12, 28]. Despite the many benefits of DL models [38], their heavy computational requirements have resulted in a significant demand for expensive and power-hungry GPUs [11]. This immense GPU demand underscores the need to *fully utilize available GPUs* to amortize costs.

DL models and workloads vary significantly in their resource usage profiles. Some DL workloads, such as BERT-based training [8], have high GPU compute requirements but a small GPU memory footprint. Others, such as Whisper-based inference [30], have a larger GPU memory footprint, but a moderate GPU compute requirement. Further, training and inference workloads have distinct resource needs, complicated by different model architectures. Even different batch sizes of a given model can result in significantly different resource requirements; for example, in our experiments, a Whisper-based inference workload with batch size of 16 had a 33% higher GPU compute utilization compared to the same workload with a batch size of 2. Consequently, *a DL workload may not fully utilize the fixed resources on a given GPU model*, resulting in underutilization of expensive GPUs.

While the variability in GPU resource requirements of DL workloads creates an underutilization problem, it also lends itself to a possible solution—*spatially sharing GPU resources among diverse DL workloads to increase GPU utilization.* Hardware support already exists for GPU spatial sharing (see Section 2). Nonetheless, there are several challenges that make it difficult to spatially share the GPU between DL workloads:

- ***Performance interference***. Sharing a GPU between just two DL workloads can result in unpredictable performance degradation for one or both colocated workloads [18, 35].

For example, when we colocated a BERT [8] training workload with an ALBERT [17] training workload, they experienced a throughput drop, relative to an exclusive run, of 40% and 37%, respectively. When the same BERT workload was colocated with a ViT [10] inference workload, the corresponding throughput drops were 19% and 3%, respectively. The much lower drop in throughput in the latter case was because the colocated Vit workload was an inference workload, and so did not require high GPU compute resources (e.g., for backpropagation). While the colocated ALBERT workload in the former case has much fewer parameters, and thus a lower memory requirement, it is a Transformer-based model, requiring significant GPU compute resources for training.

- ***Colocation candidates***. For a target DL workload that is to be executed on a GPU, there could be several candidate DL workloads for colocation (e.g., those in the ready queue of a GPU cluster service [41]). While coarse-grained resource usage patterns of individual colocation candidates can be readily obtained using an exclusive run, this information may not be enough to accurately predict the performance degradation under colocation.
- ***Profiling overhead***. A natural approach to determine which workloads to colocate is to predict their performance under colocation. Machine Learning (ML) techniques are a promising solution for such predictions. However, ML techniques typically require a large training set to make accurate predictions. For the workload colocation problem, generating multiple training samples by repeatedly running colocated workloads will require significant time and effort. Further, the time required to train complex ML models can also impose significant overheads.

In light of the above challenges, we pose our problem statement as follows: *"Given a DL workload that is to be executed, how to choose a colocated DL workload to improve GPU utilization while minimizing the performance degradation of the colocated workloads?"*

There has been some recent work on predicting colocated performance under GPU spatial sharing (see Section 6). However, such works typically focus on DL training jobs with checkpointing to save states [21, 39], and are thus not applicable to inference workloads, which are latency sensitive, or training workloads that may have performance constraints. Further, the profiling overhead can be high when predicting colocated performance. For example, multiple colocations have to be profiled online (at runtime) for each target workload to be executed [18]. There are also approaches that only rely on offline profiling, but require changes to the GPU driver or scheduler for support [35]. We thus notice *a gap in literature on solutions that achieve efficient GPU spatial sharing without hardware/firmware changes and without significant profiling overhead.*

In this work, we present KACE, a *lightweight, application-level solution* that leverages *offline profiling* to predict colocation performance under spatial sharing. KACE leverages these predictions at runtime to determine the best candidate workload to colocate with a given target workload. While KACE can generally be applied to any GPU workload, we focus on DL workloads in this work. Unlike recent works that only focus on long-running training jobs (where the profiling overhead is not important) [43], *KACE applies to both inference and training workloads.*

Using the right features, we find that a *small training set* (~20% of the entire dataset) and a simple linear regression model, with trivial training time, provide adequate prediction accuracy for KACE to effectively colocate *potentially unseen* workloads at runtime. KACE eliminates the need for extensive online profiling, as done in recent works [18], and instead relies on *one-time, offline runs* of individual workloads to extract meaningful metrics. We find that *GPU kernel metrics* provide valuable information for colocation, enabling KACE to outperform system-metric–based approaches.

We implement KACE in Python and evaluate its performance by comparing it with various baselines using 7 diverse DL workloads with different batch sizes. Our experimental results, using PyTorch for DL runs on an NVIDIA V100 GPU with MPS support, show that KACE achieves *over 90% of the colocated performance that an offline-optimal oracle policy achieves using only ~20% training data.* Compared to recent approaches, KACE provides 11% higher performance, on average, and as high as 88% for certain workloads. Compared to rule-based approaches, KACE provides 14% higher performance, on average, and as high as 52% for certain workloads. Even in the case of unseen workloads that have not been encountered in training, KACE continues to outperform other approaches, *achieving 11%–16% higher performance.* The code for KACE is publicly available to promote further research and facilitate reproducibility [1].

2 Background on GPU Sharing Mechanisms

GPU time-sharing is a resource management strategy where multiple jobs share a GPU over time (in distinct time slots), involving context switching between jobs. Various scheduling strategies, such as round-robin [24], can be employed to schedule jobs over time. However, minimizing the context switch overhead between jobs continues to be a practical challenge (see Section 6). Further, not all DL jobs can fully utilize the GPU resources [44], suggesting the need for GPU spatial sharing for improved GPU utilization.

GPU spatial sharing allows multiple jobs to execute concurrently on a GPU by partitioning GPU resources via various supported hardware and software mechanisms.

NVIDIA Multi-Instance GPU (MIG) partitions the GPU into independent resource instances of various sizes. MIG ensures isolation of compute and memory resources but cannot dynamically adjust instance sizes on the fly; reconfiguring a MIG instance and checkpoint-restarting a DL job could take minutes [18]. Further, the fixed resource sizes of MIG instances may lead to GPU underutilization due to a lack of flexibility. For example, the A100 GPU with 40GB GPU memory only supports MIG instances with memory sizes of 5GB, 10GB, 20GB, or 40GB [26]. Also, by design, MIG ensures isolation between instances, preventing the exploitation of unused GPU resources across instances. Finally, MIG is typically available only on a handful of high-end data center GPUs [26].

NVIDIA Multi-Process Service (MPS) eliminates context switches by merging colocated CUDA contexts. Specifically, MPS allocates one copy of GPU storage and schedules resources for all CUDA processes, instead of holding the context separately [6]. It also allows each process to exclusively occupy Streaming Multiprocessors (SM) via the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` variable. Due to its flexibility, MPS has been shown to outperform MIG [32]. Further, MPS is more readily available on several GPU models than MIG. We use MPS in this work for GPU spatial sharing. Note that the underutilization due to fixed-size MIG instances can be addressed by employing MPS to run multiple DL workloads within a MIG instance; in fact, MPS is supported on top of MIG [26]. As such, our solution for MPS in this work can aid MIG deployments as well.

AMD compute-unit masking in AMD GPUs allows users to specify a set of compute units on which kernels should execute [29]. This mechanism appears similar to MPS, as both enable concurrent kernel execution with memory interference. Consequently, our MPS-based solution could be applicable to AMD GPUs equipped with compute-unit masking.

3 KACE System Design and Implementation

This section describes the system design of KACE. The key components of KACE are: (1) Workload profiler, (2) Model trainer, and (3) Colocation predictor; these are depicted in Figure 1 along with the interactions between components.

3.1 Workload profiler (offline component)

The profiler collects GPU metrics for individual workloads offline. These metrics are collected one-time only.

A common approach to evaluate GPU performance is to gather overall system metrics, such as compute and memory utilization, using, for example, `nvidia_smi` [25]. However, we note that the compute and memory utilization metrics only indicate whether the GPU *is busy*, rather than providing

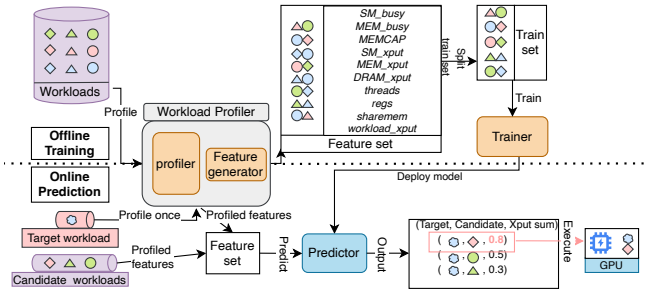

Figure 1: KACE system design showing our feature set.

actual utilization of processing units (Floating Point Process Units, Tensor cores, etc.) [43]. Thus, these metrics only provide *coarse-grained* estimates of performance.

Kernel metrics provide detailed insights into individual kernels, including metrics such as Stream Multiprocessor (SM) throughput and achieved occupancy. These metrics offer precise performance data at the kernel level, unlike `nvidia_smi` metrics which provide broader system-level information. For example, SM throughput measures the maximum throughput across 22 processing pipelines, considering operations in FP64, ALU, and FMA [27].

Despite the crucial information (*actual* resource utilization) that kernel metrics provide for interference prediction, profiling kernel metrics can take substantial time [35]. However, the profiling in KACE is done offline, and is thus not on the critical path when selecting the colocation candidate at runtime (see Section 3.3). We thus leverage both system and kernel metrics for KACE. We also include individual workload throughput (obtained via an exclusive run without colocation) as a metric as it is a strong performance indicator and can be collected with system metrics without additional profiling. The selected metrics we employ in KACE are: SM busy rate, memory busy rate, memory capacity, compute (SM) throughput, memory throughput, DRAM throughput, number of threads, number of registers, static shared memory, and throughput of single workload without colocation (shown, abbreviated, in Figure 1).

To collect metrics, we perform two offline profile runs for each workload (to prevent kernel profiling overhead from impacting system metrics). System metrics require a short run (taking, on average, ~1 minute); we execute 100 steps for each workload, excluding the first 10 steps to account for the bootstrap effect of PyTorch's lazy initialization. For the kernel metric run, we only require one step of each workload. After collecting kernel profiles, we sum up the kernel metrics across kernels and weigh them by kernel duration.

3.2 Model trainer (offline component)

To train (and test) a model for predicting the performance of colocated workloads, we run experiments where pairs of

workloads are colocated. The system design and methodology of KACE can be extended beyond two colocated workloads (see Section 5.2.5), including for non-DL workloads, since the key hurdles we have to overcome to improve colocation performance are not specific to workloads pairs or DL workloads. To fully evaluate the impact of interference, we focus on performance during the interval when both workloads are executing; as such, when any one workload completes execution, we terminate our profiling processes.

For the training data feature set, we combine the features of colocated workloads; we sum up the raw metrics (e.g., number of threads) and average out the metrics that are reported as percentages (e.g., busy rate). Finally, we normalize all features before training. While the trainer can accommodate any prediction model, we evaluate four different ML techniques in Section 5.2.

A key contribution of KACE is its ability to train with a small sample size. We evaluate prediction performance under different training set sizes, including the challenging scenario where the target workload to be colocated is not part of training, in Section 5.1; we find that KACE can predict adequately using only 20%–30% of the data for training.

3.3 Colocation predictor (online component)

The predictor determines, at runtime, the workload to colocate with a given target workload to maximize performance. While any definition of performance can be employed, we consider throughput sum to be our metric in this paper; see Section 4 for details. Given a *target* workload to be executed, the predictor iterates through all colocation *candidate* workloads (e.g., from a ready queue [41]) and predicts the throughput sum for each combination of target and candidate workloads. The candidate workload predicted to provide the highest throughput sum is selected for colocation with the target workload (see Figure 1).

3.4 Implementation

The KACE implementation is done at the application level, without any OS or hardware modifications. We implement the core logic of KACE in Python with around 2,500 lines of code. For system metric profiling, we utilize `nvidia_smi pmon` to capture the resource usage of processes [25]. For offline kernel profiling, we use `NsightCompute` [27]; `NVIDIA_MPS` is disabled in this stage. During online prediction runs, we initiate the MPS daemon. Once the predictor determines the ideal workload for colocation, it submits them to the running MPS server for colocated execution.

4 Experimental Setup and Methodology

Evaluation setup. We conduct our evaluation on a server in the Chameleon Cloud [16] equipped with 2 Intel Xeon Gold 6230 CPUs, 128GB RAM, and a 32GB NVIDIA V100 GPU. We use PyTorch 1.13 and CUDA12.3 for our experiments.

Workload	Batch	SM busy	MEM busy	MEMCAP
BERT-train [8]	2,8,16	97.0%	45.2%	5.1GB
ViT-train [10]	2,8,16	97.2%	37%	17.6GB
ALBERT-train [17]	2,8,16	97.2%	45.1%	7.1GB
BERT-inf [8]	2,8,16	95.1%	38.6%	1.4GB
ViT-inf [10]	2,8,16	28.5%	5.4%	3.2GB
Whisper-inf [30]	2,8,16	44.2%	19.6%	11.7GB
Wav2Vec2-inf [2]	2,8,16	18.9%	6.6%	12.2GB

Table 1: Workloads employed in our primary evaluation. Metrics reported in last three columns are averages across batch sizes.

Evaluation methodology. We consider a *target* workload that is scheduled for execution and a list of *candidate* workloads that can be colocated with the target workload. Our performance metric, for a given workload colocation, is the *throughput sum* of the colocated workloads, $X_1 + X_2$, where X_1 and X_2 are the throughput of the target and candidate workloads, respectively, when colocated. We focus on throughput sum as it is commonly used in colocation evaluations [35]. When reporting our results, we normalize the observed throughput sum with the throughput sum achieved by Oracle, X_{Oracle}, which is an offline-optimal policy representing the best colocated pair (described below).

$$\text{Normalized throughput sum} = \frac{X_1 + X_2}{X_{\text{Oracle}}} \tag{1}$$

Each colocation experiment is repeated 5 times. We report the average value of the normalized throughput sum along with error bars that represent the standard deviation.

Workloads. We employ 21 workloads, encompassing 7 training and inference models with 3 batch sizes each. The workloads have relatively high GPU utilization, with an average SM busy rate of over 60%. Detailed workload information is shown in Table 1. By considering all possible pairs of workloads that can be accommodated in GPU memory, we obtain 181 possible colocations. We also conduct a limited evaluation (in Section 5.2.6) with an autoregressive workload, GPT2-xl (1.5B parameters) [31], to demonstrate KACE's applicability to modern DL workloads.

Comparison baselines. We compare KACE with several baselines, including impractical ones, to evaluate its performance.

- *Oracle* is an impractical, offline-optimal policy that runs all possible colocation combinations for a given target workload and selects the pair that achieves the highest throughput sum.
- *Xu et al. [42]* developed a method to predict workload interference among co-located VMs using GPU and CPU system metrics to train a Random Forest model. Their approach involves classifying GPU kernels as long or short to estimate context switch costs, and can be applied to non-virtualized environments as well. We implement their policy by collecting the metrics mentioned in their work and training

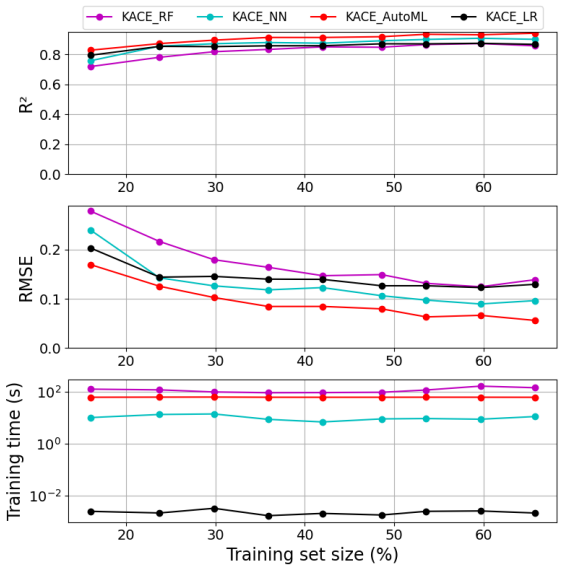

Figure 2: Prediction performance of KACE with different ML techniques as a function of training data size.

a Random Forest model to predict the throughput sum of each colocation pair.

- *Random* reports the mean of throughput sums of all possible colocations. For example, a given target workload could potentially be colocated with any of the 21 workloads we experiment with. Random obtains the throughput sum for all 21 colocations (except those that do not fit in memory) and reports the mean.
- *MEMCAP, SM%, and MEM%* are rule-based selection policies for choosing the workload to colocate with the target workload. Specifically, MEMCAP selects, for colocation with the target workload, the workload that has the smallest GPU memory footprint metric. SM% and MEM% select, for colocation, the workload that has the highest Streaming Multiprocessor (SM) utilization or the lowest GPU memory bandwidth utilization, respectively. All three metrics can be easily obtained using `nvidia-smi` [25].
- *Best rule based* picks the best-performing rule from among MEMCAP, SM%, and MEM%, for each colocation scenario. This can be considered an offline policy as it requires running all rule-based policies and selecting the best.

5 Evaluation Results

This section presents our key experimental results. We start by discussing our performance prediction results under different ML models and training set sizes. Then we present our workload colocation results, highlighting the throughput sum gain afforded by KACE compared to various baselines.

5.1 Colocation performance prediction

We evaluate the accuracy of KACE for predicting throughput sum of colocated pairs of workloads using four model candidates: Random Forest (*RF*), a 3-layer 128×64×32 Deep Neural Network (*NN*), H2O Automatic Machine Learning (*AutoML*) [13], and Linear Regression (*LR*). For *RF*, we perform a grid search, testing 100 configurations out of a search space of 4,320 to find the optimal hyperparameters. For *NN*, we apply early stopping to prevent overfitting. *AutoML* is an automated ML framework that selects optimal models from Distributed RF, Gradient Boosting, DL, and ensemble models within a given time constraint. We allocate 1 minute for *AutoML* to find the optimal model. We report prediction results on the test set for different training data set sizes.

The top two graphs in Figure 2 show the R^2 and RMSE values, respectively, for different ML techniques as a function of the training data set size (reported as a % of the total data set size). We see that the different ML techniques evaluated result in slightly different R^2 and RMSE values, with AutoML performing the best among them and Random Forst (RF) performing the worst. The superior performance of AutoML is to be expected as it sweeps through various ML techniques and employs the best one for each scenario. We find that AutoML typically picks DNN model as the best technique. The poor performance of RF is likely due to the strong correlations in the feature set, leading to a lack of diversity in decision trees and resulting in inconsistency [37]. Additionally, given the relatively small training set that is common in experimental works like ours (due to the time and effort required to generate samples), RF may result in overfitting [3].

The bottom graph of Figure 2 shows, on a log scale, the time it takes for each ML technique to be trained. Note that the training time does not change much with the training data set size as the number of training data samples is small (the total data set size is less than 200 samples). Of course, the effort required to generate more training data scales with the number of training samples. KACE_LR has the least training time, by far, among the techniques evaluated; this is to be expected as LR is a relatively simple technique. KACE_RF has the highest training time as we also perform a grid search here to look for the best hyperparameters, and RF is a more involved learning technique. KACE_AutoML also has to search through various models, and so requires time.

For the colocated workload prediction problem, a simple ML technique like ***LR performs adequately***, even outperforming more complex techniques like RF (in terms of R^2 and RMSE values), for several reasons. First, features like SM throughput may have a strong linear relationship with the throughput sum, making LR an ideal fit. Second, LR can extrapolate predictions beyond the range of training data based on regression modeling. Third, models such as LR, that have high explainability, can benefit from valuable features, such as kernel metrics, that influence the response variable (throughput sum, in our case).

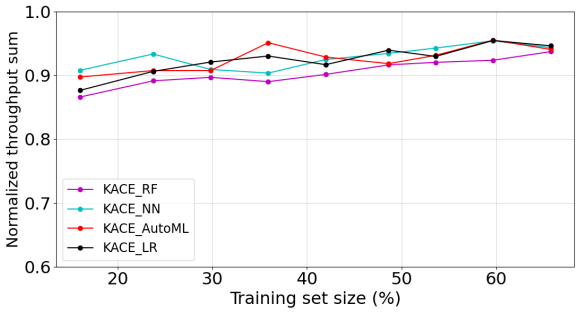

Figure 3: Normalized throughput sum achieved by KACE under different ML techniques.

5.2 Workload colocation results

We now present our workload colocation results for KACE.

5.2.1 KACE under different ML techniques. Figure 3 shows the normalized throughput sum achieved by KACE, averaged over all 21 target workloads in the test set, using different ML techniques. We see that the throughput sum achieved gradually increases with the training set size as the model learning improves with more training data. Interestingly, the achieved throughput sum values are quite similar across ML techniques; this is in agreement with the somewhat similar R^2 and RMSE values obtained by the different ML techniques in Figure 2. Given the *low training effort for LR and its ability to obtain competitive throughput results under colocation*, we choose LR as the prediction model for KACE.

5.2.2 KACE vs. other baseline policies. We now compare KACE with other baseline policies to evaluate the colocation performance. In real-world scenarios, significant training data may not be available or might require substantial time and effort to generate. As such, we consider a small training data set size of ~20% of the entire dataset (corresponding to about 40 training samples). Results did not qualitatively change with larger training data set sizes.

Figure 4 shows the normalized throughput sum achieved by KACE (using LR) and other baseline policies, averaged over all 21 test set workloads. The normalization for each test case is with respect to Oracle; as such, Oracle shows a value of 1. Starting with Random, we see that it only achieves 54% of the throughput sum achieved by the offline-optimal Oracle. This is not surprising as Random effectively colocates a given target workload with a random workload, which can result in significant GPU contention and performance degradation for the colocated workloads.

The rule-based policies, MEMCAP, SM%, and MEM%, result in a wide range of results, with SM% performing even worse than Random, and MEMCAP and MEM% achieving 70%–80% of the performance gains afforded by Oracle. Recall that MEMCAP selects the workload for colocation that has the smallest GPU memory footprint. However, this metric

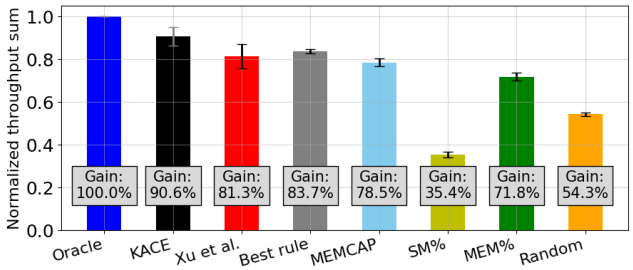

Figure 4: Normalized throughput sum of different colocation policies when using a small training data set.

provides limited information and cannot capture, for example, the model architecture details. SM% selects the workload for colocation that has the highest SM utilization; while this should maximize GPU utilization under colocation, it can lead to high resource contention. MEM% selects the colocation workload that has the smallest memory busy rate, favoring workloads with fewer data transfers and smaller input data. As such, this policy routinely overlooks colocation candidates such as speech recognition inference that perform frequent data transfers.

The Best rule-based policy selects the best of the three rules (MEMCAP, SM%, and MEM%) for each test set target workload, and as such has a higher throughput sum than the individual rule-based policies. While the Best rule-based policy is impractical, it serves as a good baseline policy. For example, we note that Best rule-based improves upon MEMCAP by achieving a roughly 7% higher average throughout sum of 83.7%. This suggests that MEMCAP likely outperforms the other rule-based policies (SM% and MEM%) in individual test cases; on further inspection, we find that MEMCAP is the best single rule-based policy in 13 out of the 21 test scenarios.

Xu et al. [42] achieves 81% of the throughput sum achieved by Oracle. This is slightly higher than that achieved by MEMCAP, but lower than that achieved by the (impractical) Best rule-based policy. Given that the feature set of Xu et al. is more extensive than the singular feature employed by MEMCAP, the above results suggest that the additional features in Xu et al. do not provide significant benefits. This highlights *the importance of selecting useful features when predicting colocated performance under GPU sharing.*

KACE outperforms all other comparison baselines, ***achieving close to 91% of the throughput sum achieved by Oracle***. Compared to the next-best practical policy (since Best rule-based requires offline runs), Xu et al., ***KACE achieves an 11% higher average throughput sum*** (relatively speaking). The superior performance of KACE can be attributed to the *kernel information* that it employs as part of its feature set, which allows it to better estimate GPU resource contention. We note that Xu et al. also employ some kernel metrics, such

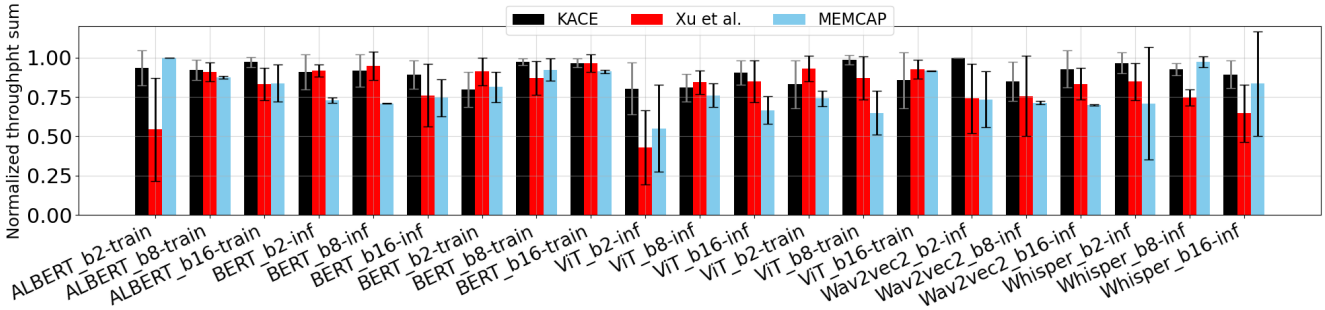

Figure 5: Normalized throughput sum of KACE, Xu et al., and MEMCAP (best rule-based policy) for all test cases.

as long/short kernel information, but these features aim to capture GPU context switches among VMs, which is different from our objective of maximizing throughput.

5.2.3 Results for individual workloads. To better understand the performance of KACE on individual test workloads, we plot the normalized throughput sum achieved by KACE, Xu et al., and MEMCAP for all 21 test set workload colocations in Figure 5; error bars represent the standard deviation over the 5 runs for each colocation. The '_bx' notation in the workload name refers to the batch size of x and the '-train'/'-inf' refers to the workload type. We do not show other policies as they are either impractical or perform worse, and to aid comparisons. Of the 21 colocations, we find that KACE is the best policy in 13 cases. In these 13 cases, *KACE outperforms Xu et al. and MEMCAP by 20% and 24%, respectively, on average, and by as much as 88% and 52%, respectively.* Of the remaining 8 cases, Xu et al. is the best policy in 6 of them, and outperforms KACE by 7%, on average, and by as much as 14%. In 2 cases, MEMCAP is the best policy, and outperforms KACE by 6%, on average, and by as much as 7%.

We see that policy performance varies with different workloads. For example, both Xu et al. and MEMCAP underperform in the case of ViT inference, especially for batch size 2 ('ViT_b2-inf'). On further inspection, we find that Xu et al. consistently selects ViT_b2-inf as the colocation candidate due to its low SM and MEM usage, underestimating the throughput benefits of larger batch sizes, and thus fails to choose the optimal ViT_inf-b16 colocation candidate. In contrast, KACE additionally considers individual workload throughput, thereby often selecting the optimal colocation candidate. On the other hand, MEMCAP frequently chooses BERT_b2-inf as the colocation candidate due to its minimal memory footprint, but its high compute needs (92% SM busy rate) lead to significant interference when paired with ViT_b2-inf, which has moderate compute needs (29% SM busy rate). KACE accounts for thread and register usage, incorporating GPU compute information, and thus outperforms MEMCAP. A similar behavior is also observed when colocating with the Whisper inference workload.

5.2.4 Results for unseen workloads. We now consider the challenging case where the target workload to be executed is not part of the training set (under any batch size); this could arise in practice either because a new workload is encountered at runtime or because the training set size is limited. We evaluated all 7 distinct workloads with the highest batch size, ensuring that the target workload, for any batch size, is not included in training. We find that KACE continues to perform well, achieving ~92% of the throughput sum achieved by Oracle. Compared to Xu et al. and rule-based policies, KACE achieves 11% and 16% higher average throughput sum, respectively. The above results show that ***KACE performs well even for unseen workloads***.

5.2.5 Results for multiple unseen workloads. We now demonstrate KACE's ability to handle multiple (more than 2) colocated workloads; we consider the unseen workload case. Specifically, we evaluate two colocation configurations: (1) a target, unseen workload to be colocated with two other candidate workloads; and (2) a target, unseen workload to be colocated with three other candidate workloads. Our experimental results (not included here) show that KACE continues to perform well, achieving 96% and 91% of the throughput sum compared to Oracle when colocating with two and three workloads, respectively. While KACE can be extended beyond four colocated workloads, as the size of deep learning models continues to increase, we do not expect that numerous workloads will be needed to saturate a GPU while maintaining performance efficiency [4].

5.2.6 Results for an unseen autoregressive workload. We now consider colocation with an autoregressive workload, given that such modern models can be different from traditional vision and NLP models. Specifically, we experiment with the GPT2-xl model [31] with output token lengths of 10, 20, and 214 (the average token length in LMSYS-CHAT-1M dataset [22]). To maintain low profiling overhead, we profile

kernel metrics during the prefill stage and the decoding stage of the first token (representative of compute-light sequential token generation stages); the MEMCAP system metric captures the memory footprint of the entire decode stage as the MEMCAP value increases with output token length. We use the same training dataset as in Table 1 (that is, without including GPT2-xl) and test with GPT2-xl as the target, unseen workload to be colocated with another workload. We find that KACE achieves 92.3% of the throughput sum compared to Oracle and is ∼10% better than the next-best policy (Best rule-based). Xu et al. performs poorly in this case (57.8% of Oracle) as it does not consider the total memory footprint, an important feature for autoregressive models given their large output token length. The above results suggest that KACE can be extended to modern GPU workloads as well.

6 Related Work

Performance prediction under colocation. Colocation performance predictions can be made using either online profiling in the presence of interference or offline profiling based on monitored metrics. While online profiling offers high accuracy, it cannot proactively schedule workloads for colocation, which is the focus of our work.

Xu et al. [42] predict interference between colocated DL jobs that are run on VMs on the same GPU. The authors also focus on predicting the VM context switch costs, and so select features for training that reflect these costs. As discussed in Section 5.2, KACE outperforms Xu et al. due to its more appropriate feature set. MISO [18] predicts the optimal MIG partitions by evaluating colocation speedup with MPS, requiring multiple online profiles for each colocation to determine the best configuration. Horus [43] uses DL computation graphs during offline profiling to predict GPU utilization when colocating DL training workloads. However, its objective of maximizing GPU utilization need not translate to maximizing throughput sum, as acknowledged by the authors of Horus. Further, as shown using the SM% approach in Figure 4, maximizing SM utilization can lead to poor throughput gains due to high interference. Finally, both Horus and MISO are primarily suited for training workloads due to checkpoints and offline runs, which are impractical for inference jobs with performance requirements.

We note that there are works that focus on performance prediction under colocation [7, 15, 23, 36], but for non-GPU workloads. As such, these works cannot predict the colocated performance of GPU-based DL workloads as they are oblivious to GPU features.

GPU scheduling for DL workloads. Prior GPU scheduling works involving temporal sharing aim to minimize the cost of state swapping between DL jobs. Gandiva [40] introduces a checkpoint-restart mechanism to transfer training states between the GPU and host memory. Salus [44] optimizes GPU state management to reduce context switches. There are also other works that focus on inference serving with temporal sharing [33, 34]. KACE does not focus on temporal sharing of the GPU, and instead considers spatial sharing.

For spatial sharing of the GPU, Wavelet [39] and Zico [21] colocate forward and backward passes of training tasks to reduce peak memory usage. IADeep [4] predicts training-specific metrics when colocating and exploits long-running training jobs to stop/restart them for optimization. However, these solutions are specific to training workloads. GSLICE [9] and gpulet [5] focus on performance isolation for colocated GPU workloads with the objective being predictable performance, which is different from KACE's objective of maximizing throughput. SHEPHERD [45] and AlpaServe [20] improve GPU efficiency for model serving, but do not primarily focus on interference-aware GPU colocation. Orion [35] reduces GPU interference via kernel-level scheduling through the colocation of memory- and compute-intensive kernels. However, Orion requires offline kernel labeling and specific CUDA support for effective scheduling. In contrast, KACE focuses on workload-level solutions, tackling GPU interference from the application perspective. Further, Orion aims to minimize interference among a given colocated workload set whereas KACE aims to find workloads to colocate to maximize throughput. As such, KACE and Orion are somewhat complementary. Similarly, KACE can be considered complementary to (the AMD-specific) REEF [14].

7 Conclusion

This paper presents KACE, a framework for efficient GPU spatial sharing that accurately and quickly predicts interference among colocated DL workloads. Our key contributions include: (i) selecting a set of kernel and system metrics that provide valuable information to predict colocated performance; (ii) using limited and one-time offline and exclusive profiling of individual workloads, eliminating the need for costly online profiling; (iii) identifying a simple ML model that provides adequate colocation performance prediction accuracy with minimal training time and a small training dataset; and (iv) experimentally evaluating KACE over multiple training and inference workloads and against various baselines. Evaluation results show that KACE achieves over 90% of the throughput sum achieved by Oracle using only a fraction of the training data.

Acknowledgment

We thank the anonymous reviewers and our shepherd, George Neville-Neil, for providing their valuable feedback. This work was supported by NSF grants CCF-2324859, CNS-2214980, CNS-2106434, CNS-1750109, CNS-2106027, CNS-2146909, and CCF-2046444.

References

[1] 2024. KACE Artifact. https://github.com/nba556677go/KACE-artifact GitHub repository.

[2] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. arXiv:2006.11477 [cs.CL] https://arxiv.org/abs/2006.11477

[3] Lasai Barreñada, Paula Dhiman, Dirk Timmerman, Anne-Laure Boulesteix, and Ben Van Calster. 2024. Understanding random forests and overfitting: a visualization and simulation study. arXiv:2402.18612 [stat.ME] https://arxiv.org/abs/2402.18612

[4] Wenyan Chen, Zizhao Mo, Huanle Xu, Kejiang Ye, and Chengzhong Xu. 2023. Interference-aware Multiplexing for Deep Learning in GPU Clusters: A Middleware Approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*. Denver, CO, USA.

[5] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[6] NVIDIA Corporation. 2021. *CUDA Multi-Process Service Overview*. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[7] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. Houston, TX, USA, 77–88.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] https://arxiv.org/abs/1810.04805

[9] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 492–506. https://doi.org/10.1145/3419111.3421284

[10] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. arXiv:2010.11929 [cs.CV] https://arxiv.org/abs/2010.11929

[11] Hugging Face. 2023. The Llama 3 Models Were Trained on 15 Trillion Tokens with 24,000 GPUs. https://huggingface.co/blog/llama3#:~:text=The%20Llama%203%20models%20were,two%20clusters%20with%2024%2C000%20GPUs. Accessed: 2024-07-03.

[12] GitHub. [n. d.]. GitHub Copilot. https://copilot.github.com/.

[13] H2O.ai. 2024. *H2O.ai AutoML Documentation*. https://docs.h2o.ai/h2o/latest-stable/h2o-docs/index.html.

[14] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[15] Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. 2019. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *Proceedings of the 10th ACM Symposium on Cloud Computing (SOCC '19)*. Santa Cruz, CA, USA.

[16] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[17] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. arXiv:1909.11942 [cs.CL] https://arxiv.org/abs/1909.11942

[18] 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 the 13th Symposium on Cloud Computing (SoCC '22)*. San Francisco, CA, USA, 173–189.

[19] Nan Li, Karthik Shankar, Zhigang Tang, and Amit Lahiri. 2021. Artificial Intelligence for Radiographic COVID-19 Detection: A Systematic Review. *Data Science and Management* 2, 6 (2021), 100218. https://doi.org/10.1016/j.patter.2021.100218

[20] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[21] Gangmuk Lim, Jeongseob Ahn, Wencong Xiao, Youngjin Kwon, and Myeongjae Jeon. 2021. Zico: Efficient GPU Memory Sharing for Concurrent DNN Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 161–175. https://www.usenix.org/conference/atc21/presentation/lim

[22] LMSYS Team. 2023. LMSYS-CHAT-1M: A Dataset for Large-Scale Multimodal System Chat. Online. https://arxiv.org/abs/2309.11998 Accessed: YYYY-MM-DD.

[23] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 248–259.

[24] NVIDIA. 2021. NVIDIA AI Enterprise: Deployment Guide on VMware. https://docs.nvidia.com/ai-enterprise/deployment-guide-vmware/0.1.0/advance-gpu.html Accessed: 2024-07-03.

[25] NVIDIA Corporation. 2016. *NVIDIA System Management Interface*. https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf Version 367.38.

[26] NVIDIA Corporation. 2021. *NVIDIA Multi-Instance GPU User Guide*. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html Accessed: 2024-07-03.

[27] NVIDIA Corporation. 2023. *NVIDIA Nsight Compute*. https://developer.nvidia.com/nsight-compute.

[28] OpenAI. [n. d.]. ChatGPT. https://openai.com/chatgpt/.

[29] Nathan Otterness and James H. Anderson. 2021. Exploring AMD GPU Scheduling Details by Experimenting With "Worst Practices". In *Proceedings of the 29th International Conference on Real-Time Networks and Systems* (NANTES, France) *(RTNS '21)*. Association for Computing Machinery, New York, NY, USA, 24–34. https://doi.org/10.1145/3453417.3453432

[30] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2022. Robust Speech Recognition via Large-Scale Weak Supervision. arXiv:2212.04356 [eess.AS] https://arxiv.org/abs/2212.04356

[31] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. https://api.semanticscholar.org/CorpusID:160025533

[32] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. 2023. An Analysis of Collocation on GPUs for Deep Learning Training. arXiv:2209.06018 [cs.LG] https://arxiv.org/abs/2209.06018

[33] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[34] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[35] Foteini Strati, Xianzhe Ma, and Ana Klimovic. 2024. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys '24)*. Athens, Greece, 1075–1092.

[36] Amoghavarsha Suresh and Anshul Gandhi. 2021. ServerMore: Opportunistic Execution of Serverless Functions in the Cloud. In *Proceedings of the 2021 ACM Symposium on Cloud Computing (SoCC '21)*. Seattle, WA, USA, 570–584.

[37] Cheng Tang, Damien Garreau, and Ulrike von Luxburg. 2018. When do random forests fail?. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/204da255aea2cd4a75ace6018fad6b4d-Paper.pdf

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[39] Guanhua Wang, Kehan Wang, Kenan Jiang, XIANGJUN LI, and Ion Stoica. 2021. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 696–710. https://proceedings.mlsys.org/paper_files/paper/2021/file/099268c3121d49937a67a052c51f865d-Paper.pdf

[40] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[41] 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 *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 533–548. https://www.usenix.org/conference/osdi20/presentation/xiao

[42] Xin Xu, Na Zhang, Michael Cui, Michael He, and Ridhi Surana. 2019. Characterization and Prediction of Performance Interference on Mediated Passthrough GPUs for Interference-aware Scheduler. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA, USA.

[43] Gingfung Yeung, Damian Borowiec, Renyu Yang, Adrian Friday, Richard Harper, and Peter Garraghan. 2022. Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 1 (2022), 88–100. https://doi.org/10.1109/TPDS.2021.3079202

[44] Peifeng Yu and Mosharaf Chowdhury. 2019. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. *CoRR* abs/1902.04610 (2019). arXiv:1902.04610 http://arxiv.org/abs/1902.04610

[45] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. 2023. SHEPHERD: Serving DNNs in the Wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 787–808. https://www.usenix.org/conference/nsdi23/presentation/zhang-hong