Predicting workload colocations under GPU spatial sharing

RPE Report

Bing-Shiun Han

Department of Computer Science
Stony Brook University

This report is submitted for the RPE exam requirement of
Doctor of Philosophy

October 2024

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this report are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This report is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text.

Bing-Shiun Han
October 2024

Abstract

The increasing computation demand for deep learning workloads has driven the need for efficient GPU sharing. GPU spatial sharing among workloads is an effective approach to increase resource utilization and reduce the monetary and environmental costs of running deep learning workloads. Spatial sharing allows multiple workloads to execute concurrently on a GPU by partitioning GPU resources via various supported hardware and software mechanisms. Common spatial sharing techniques, such as NVIDIA MPS and NVIDIA MIG, achieve performance isolation by partitioning compute or memory resources for individual workloads. However, managing GPU resources across multiple colocated workloads presents significant challenges, particularly performance degradation due to resource contention. Existing approaches to mitigate interference often require extensive profiling of all colocation candidates, making them impractical for deployment.

In this RPE, we propose a lightweight, prediction-based approach to effectively colocate workloads on a spatially shared GPU. We use NVIDIA MPS, a commonly used spatial sharing mechanism that partitions GPU Streaming Multiprocessors (SM) to achieve compute isolation, as our framework. We test our solution on 7 commonly used deep learning training and inference workloads, and accurately predict colocation interference using exclusive kernel metrics with limited training data and minimal training time, eliminating the need for extensive online profiling. Experimental results show our method outperforms existing rule-based and prediction-based policies by 16% and 10%, respectively, and achieves performance within 10% of an offline-optimal oracle policy. As future work, we plan to extend our solution beyond pairs of colocated workloads.

Table of contents

Chapter 1

Introduction

Applications of Deep Learning (DL) models, such as image processing and speech recognition, have exponentially grown in recent years [20], resulting in useful services and products [9, 30]. Despite the many benefits of DL models [39], their heavy computational requirements have resulted in a significant demand for expensive and power-hungry GPUs [8]. 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 [5], have high GPU compute requirements but a small GPU memory footprint. Others, such as Whisper-based inference [34], have a larger GPU memory footprint, but a moderate GPU compute requirement. Likewise, their GPU I/O requirements can vary as well. Further, training and inference workloads have distinct resource needs, further 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, available resources on a given GPU model*, resulting in underutilization of expensive and much needed 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 Chapter 2). Nonetheless, there are several challenges that make it difficult to spatially share the GPU between colocated DL workloads:

- ***Performance interference***. Sharing a GPU between just two DL workloads can result in unpredictable performance degradation for one or both colocated workloads [36, 19]. For example, when we colocated a BERT [5] 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 [6] 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, which requires 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 [45]). 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 Chapter 3). However, such works typically focus on DL training jobs with checkpointing to save states [40, 21], 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 [19]. There are also approaches that only rely on offline profiling, but require changes to the GPU driver or scheduler for support [36]. 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 Research Proficiency Examination report, we first introduce the common GPU sharing concepts and performance metrics. Furthermore, we dive into the related works on predicting workload colocations and scheduling DL workloads under GPU sharing. To

evaluate our thoughts as a prototype, we present KACE (**K**ernel-**A**ware **C**olocation for **E**fficient GPU spatial sharing), 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) [47], *KACE applies to both inference and training workloads*.

Using the right features, we find that a *small training set* ($\sim$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 [19], 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 $\sim$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*

1.1 Chapter organization

The rest of this report is organized as follows. Chapter 2 provides the necessary background on how GPU sharing works and characteristics of DL workloads. Chapter 3 summarizes some of the works in the areas of predicting colcations and GPU scheduling. Chapter 4 discusses the system design of KACE. Chapter 5 discuss the evaluation methodology and comparison baselines. Chapter 6 demonstrate the evaluation results of KACE, and we conclude the report with future works in Chapter 7.

Chapter 2

Background

In this chapter, we explain how deep learning tasks function and how GPU scheduling flow is designed. We also examine the common metrics used to measure GPU performance. Lastly, we discuss the typical methods GPUs use to share resources and enhance efficiency.

2.1 Characteristics of Deep Learning workloads

Deep Learning (DL) workloads can be broadly categorized into training and inference workloads, each with distinct characteristics and resource requirements. DL training workloads involve iterative processes where models learn from large datasets through multiple epochs. Training involves two main phases: forward propagation and backward propagation. In forward propagation, input data passes through the neural network to produce predictions. During backward propagation, the model's predictions are compared to the actual outcomes, and the errors are used to adjust the model's weights through gradient descent, improving accuracy over time. Since backward propagation requires updating billions of parameters, the training process is resource-intensive, typically exhibiting high GPU utilization and periodic memory access patterns due to the repeated processing of data batches.

In contrast, DL inference workloads are focused on deploying trained models to make predictions or classifications on new data. Inference tasks generally have lower GPU utilization since it only involves forward propagation to generate predictions. Also, their resource usage depends heavily on request frequencies, thus the resource usage pattern largely depends on incoming request rates. While training requires significant computational power and memory bandwidth to handle extensive computations and data transfers, inference workloads demand quick response times and often need to maintain Service Level Agreements (SLA) to provide reliable services.

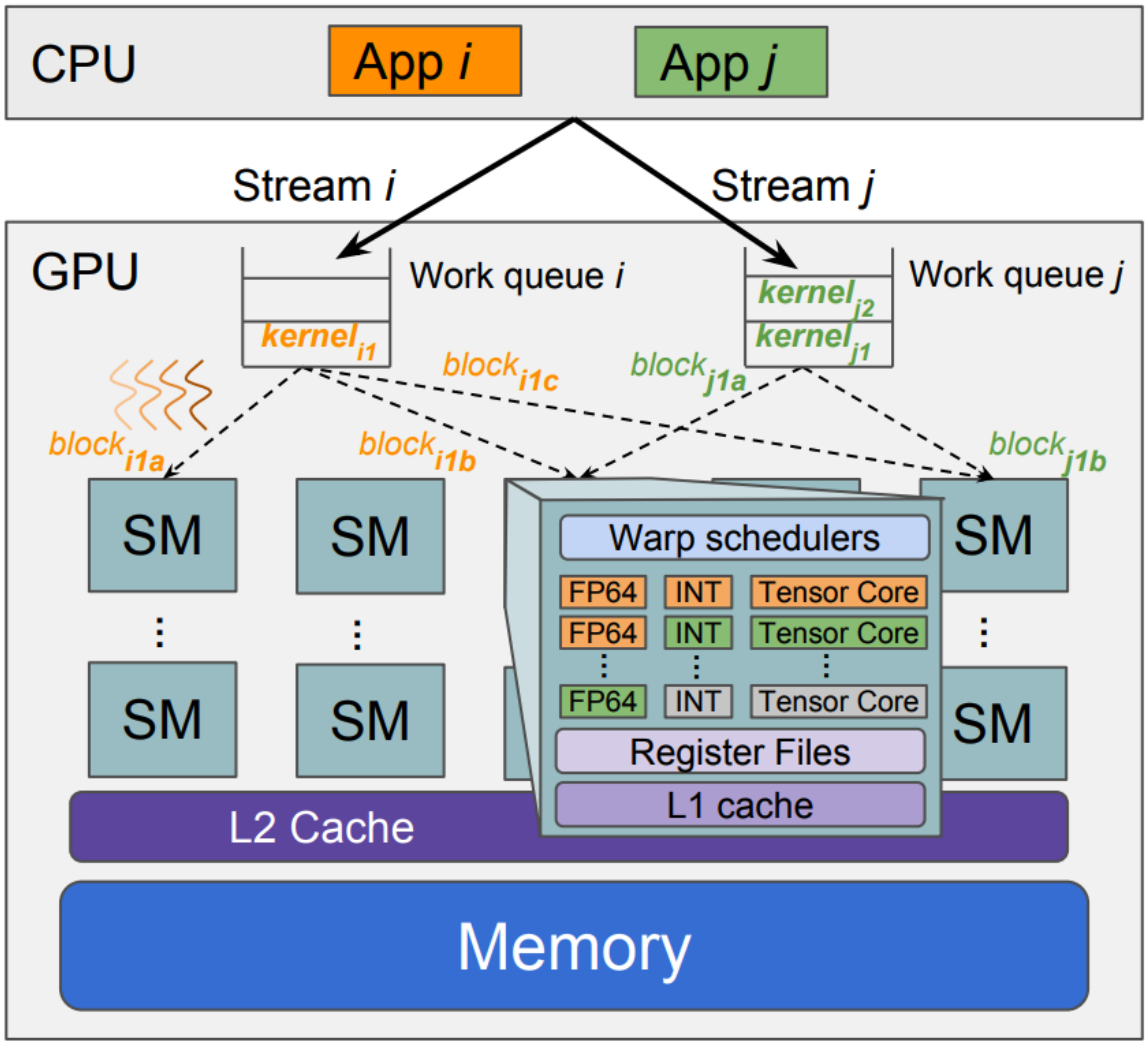

Fig. 2.1 simplified GPU architecture. source - Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications [36]

2.2 GPU architecture

In this chapter, we briefly explain GPU architecture and scheduling flow. A simple GPU architecture is shown in Figure 2.1. A GPU consists of multiple Streaming Multiprocessors (SMs), each containing various types of compute cores (e.g., FP64 units, tensor cores, Floating Point Multiply and Accumulate Units (FMA), [31]), memory load/store units, register files, and L1 cache. There are also shared L2 cache and memory across SMs. In short, an SM can be considered the "CPU core" in a GPU, serving as the basic computation unit.

2.2.1 GPU programming abstraction

When a DL application, written in PyTorch or TensorFlow, is launched, its operations (e.g., convolution, batch normalization) are compiled and submitted to the GPU as CUDA computation kernels. CUDA is a software layer designed by NVIDIA that provides direct access to the GPU's virtual instruction set and parallel computational elements for executing compute kernels [42]. A GPU kernel, unlike an OS kernel, is a function called from a host program to perform calculations or other operations on the GPU. Submitting a kernel involves specifying its resource requirements (e.g., the number of thread blocks, registers, threads per block, and shared memory required), which vary among CUDA versions. Threads are packaged in blocks, so block size have affect on thread GPU placement. The application associates each kernel launch and memory operation with a particular CUDA stream. A stream is a sequence of operations guaranteed to execute in order. Each application process has its own default stream, but to increase concurrency, applications can create additional streams and submit kernels across them [36].

2.2.2 GPU scheduling flow

As shown in Figure 2.1, the GPU buffers each CUDA stream's kernels in a separate work queue on the device. Most GPUs, particularly NVIDIA GPUs, do not allow users to preempt kernels. The GPU hardware scheduler dispatches thread blocks from kernels in each work queue based on stream priority. The scheduler assigns a thread block to an SM when the thread block's data dependencies are met and an SM with sufficient resources is available. Users cannot control which SM will execute a particular thread block, a detail not disclosed by NVIDIA. When a thread block is assigned to an SM, the SM will schedule and execute all thread warps from that block. SMs can execute multiple warps concurrently, from thread blocks that may belong to different kernels and streams [24]. However, if any warp saturates a resource on the SM (such as the number of registers), the SM's warp scheduler will wait until no resource is saturated before scheduling any additional warps, even if compute unit or shared memory on the SM are available [36].

2.3 GPU performance metrics

In this chapter, we discuss the common GPU metrics that are collected as resource profiles.

#Date	Time	gpu	pid	type	sm	mem	enc	dec	command
#YYYYMMDD	HH:MM:SS	Idx	#	C/G	%	%	%	%	name
20240630	16:47:10	1	486919	C	-	-	-	-	nvidia-cuda-mps
20240630	16:47:11	1	486919	C	-	-	-	-	nvidia-cuda-mps
20240630	16:47:12	1	486919	C	-	-	-	-	nvidia-cuda-mps
20240630	16:47:13	1	486919	C	-	-	-	-	nvidia-cuda-mps
20240630	16:47:14	1	486919	C	-	-	-	-	nvidia-cuda-mps
20240630	16:47:15	1	486919	C	-	-	-	-	nvidia-cuda-mps
20240630	16:47:16	1	486919	C	9	0	-	-	nvidia-cuda-mps
20240630	16:47:16	1	1518561	C	-	-	-	-	python
20240630	16:47:17	1	486919	C	19	3	-	-	nvidia-cuda-mps
20240630	16:47:17	1	1518554	M+C	-	-	-	-	python
20240630	16:47:17	1	1518561	M+C	-	-	-	-	python
20240630	16:47:18	1	486919	C	67	29	-	-	nvidia-cuda-mps
20240630	16:47:18	1	1518554	M+C	-	-	-	-	python
20240630	16:47:18	1	1518561	M+C	-	-	-	-	python
20240630	16:47:19	1	486919	C	98	47	-	-	nvidia-cuda-mps
20240630	16:47:19	1	1518554	M+C	-	-	-	-	python
20240630	16:47:19	1	1518561	M+C	-	-	-	-	python
20240630	16:47:20	1	486919	C	99	46	-	-	nvidia-cuda-mps

Fig. 2.2 sample output of `nvidia_smi pmon`

2.3.1 Overall GPU metrics

A common approach to evaluating GPU performance is to gather overall system metrics, such as compute and memory utilization, using tools like `nvidia_smi pmon` [27]. Sample output is shown in Figure 2.2. Common metrics include:

- **sm%** - Also known as SM busy rate. This metric represents the percentage of time over the past sample period during which one or more kernels were executed on the GPU, often referred to as GPU utilization.
- **mem%** - Also known as GPU memory busy rate. This metric indicates the percentage of time over the past sample period during which device memory was being read or written, often referred to as GPU memory utilization.
- **enc%** - This metric shows the percentage of time over the past sample period during which the video encoder was being used.

- **dec%** - This metric reflects the percentage of time over the past sample period during which the video decoder was being used.

It is important to note that the compute and memory busy rate metrics only indicate whether the GPU *is busy*, rather than providing actual utilization of processing units (Floating Point Units, Tensor Cores, etc.) [47]. Therefore, these metrics only provide *coarse-grained* estimates of performance. For example, a GPU workload reporting 100% utilization implies that kernels are executing continuously, the compute units may still be significantly underutilized.

Another common metric is memory capacity, also known as memory usage, which reflects how much GPU memory is reserved by a GPU process. Once memory usage reaches the GPU limit, an Out Of Memory (OOM) error is triggered, and all processes will be forcefully terminated. This is a major constraint for accommodating DL workloads.

2.3.2 Kernel metrics

Kernel metrics offer precise performance data at the kernel level, unlike `nvidia_smi` metrics which provide broader system-level information. Kernel metrics can be collected from `Nsight_Compute`, an official profiling tool provided by NVIDIA. Common metrics include [32]:

- **Stream Multiprocessor (SM) throughput** - Measures the maximum throughput across 22 processing pipelines, considering operations in FP64 and FMA.
- **GPU memory throughput** - Measures the maximum memory access throughput and requests across 21 metrics. The measured devices mostly include the L1 data cache for SM, two parallel pipelines (including the load/store unit), and TEX for texture lookups and filtering.
- **Registers per thread** - The number of registers used for each thread.
- **Block size** - The number of threads in a block.
- **Shared Memory** - Shared GPU memory is a type of virtual RAM reserved from the actual RAM of the computer itself. Shared memory is a partition of RAM that connects to the GPU using PCIe connectors, thus the speed is slower than GPU device memory [7].

- **Achieved occupancy** - The number of active warps varies over the duration of the kernel. Maintaining as many active warps as possible (high occupancy) helps avoid situations where all warps are stalled and no instructions are issued. Achieved occupancy is measured on each warp scheduler using hardware performance counters to count the number of active warps on that scheduler every clock cycle. These counts are then summed across all warp schedulers on each SM and divided by the clock cycles the SM is active to find the average active warps per SM. Low achieved occupancy can indicate an unbalanced workload within or across blocks, too few blocks launched, or a partial last wave [26].

2.4 GPU temporal sharing

GPU temporal sharing is a resource management strategy where multiple jobs share a GPU over time (in distinct time slots), involving context switching between jobs. Cost of GPU Context switch is 25-50 micro seconds. Various scheduling strategies, such as round-robin [25], can be employed to schedule jobs over time. However, minimizing the context switch overhead between jobs continues to be a practical challenge (see Section 3). Further, since most DL jobs can not fully utilize the GPU resources [48], GPU spatial sharing is more preferred to improve GPU utilization.

2.5 GPU spatial sharing

GPU spatial sharing allows multiple jobs to execute concurrently on a GPU by partitioning GPU resources via various supported hardware and software mechanisms. Below we discuss the popular GPU spatial sharing approaches that exist, including the one we employ in this work.

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 [19]. 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 [28]. Finally, MIG is typically available only on a handful of high-end data center GPUs [28].

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 [4]. 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 [35]. 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 [28]. As such, our solution for MPS in this work can aid MIG deployments as well.

Chapter 3

Related Work

Our work focuses on prediction-based GPU scheduling. Therefore, we divide the related work into two sections. First, we review how previous works predict the performance of DL and non-DL workloads. Secondly, we examine how previous works improve GPU scheduling under different sharing schemes.

3.1 Performance prediction under colocations

Predicting colocation performance often requires resource profiles to build accurate prediction models. These profiles are derived from the dominant device, which is GPU for DL workloads, and CPU/memory for cloud and web applications. We explore these two sub-categories in the following sections.

3.1.1 Colocations for DL workloads

Colocation performance predictions can be made using either offline profiling based on GPU system and kernel metrics or online profiling with dry runs. While online profiling offers high accuracy, it cannot proactively schedule workloads for colocation, which is why we choose offline profiling for our work.

Xu et al. [46] predict interference between colocated DL jobs on temporal-shared VMs using individual kernel and system profiles of each workload. This work highlights the significance of kernel length, discovering that short kernels are blocked by long kernels (>1ms) when switched out to CPUs. This results from the Best Effort scheduler used for NVIDIA vGPUs, which employs round-robin to share resources based on actual demand. By incorporating long and short kernel lengths alongside overall GPU utilization and PCIe

bandwidth as features, this work achieves a Mean Square Error of 0.151 using a trained random forest model.

However, this solution cannot be directly applied to spatial sharing due to different working principles. For instance, MPS merges all CUDA scheduling contexts into one, so interference arises from sharing L1/L2 cache, which is a finer-grain issue than context switches, making kernel length less impactful on interference. In comparison, our solution, KACE, outperforms Xu et al. due to its more suitable feature set, as discussed in Chapter 6.2.

Horus [47] extracts DL computation graph metrics (e.g., MaxPool, Conv, Relu counts), FLOPs, and memory parameters to predict GPU utilization for colocated workloads using XGBoost. To estimate interference for over-committing utilizations, polynomial fitting is employed to predict job slowdowns more accurately. The predicted utilization and memory requirements are then used to calculate the scheduling cost of a job and categorize waiting jobs by resource needs. Horus achieves 61.5% GPU utilization and reduces job wait time by 68.3%.

However, this method is not applicable to inferencing, as utilization is not as predictable as in training. While DL graph metrics are useful for accurately predicting memory estimations due to parameter information, actual utilization also depends on data congestion rates, which are influenced by request rates. The study indicates that system and kernel metrics have higher significance than NN graph metrics for utilization prediction. The top features identified are FLOPS, memory parameters, and batch size, which can be obtained from the model information card and overall system metrics. In contrast, KACE retains kernel profiling to achieve fine-grained model information applicable to both training and inference, without relying on specific DL graph information.

MISO [19] predicts optimal MIG partitions by evaluating colocation speedups with MPS. It requires three online profile runs per colocation to determine the slowdown rate under zero, light, and heavy interference by adjusting the threads for each workload. These multiple profiles with different MPS thread allocations enable interference prediction under varying sharing levels. The MPS profile approach avoids hardware restart overheads (seconds to minutes) when changing MIG partitions. MISO achieves 49% and 16% lower average job completion times compared to unpartitioned and optimal static GPU partition schemes, respectively.

However, the profile overhead of running three dry runs for each colocation can be a bottleneck. While MISO claims to complete each run within seconds due to the constant progression rate of training, this does not apply to real-time inference tasks with strict latency constraints. Additionally, the first few training steps include model initialization times, which can result in incorrect profiled completion times.

3.1.2 Colocations for non-DL workloads

Several works focus on performance prediction under colocation for non-GPU workloads, mostly for cloud applications. However, these approaches are not suitable for predicting the colocated performance of GPU-based DL workloads as they do not consider GPU features.

Servermore [37] optimizes CPU, memory, and Last Level Cache (LLC) to enhance colocation with latency-sensitive, short-lived serverless jobs alongside serverful VMs. Bubble-up [22] forecasts colocation interference in data center memory systems by calculating pressure scores with pre-measured memory pressure curves for each workload. This approach improves memory utilization by 50-90% while maintaining a prediction error of just 1%.

Scavenger [14] is a black-box solution that colocates latency-sensitive workloads in a public cloud with batch jobs by regulating processor usage, memory capacity, and network bandwidth. It aggressively throttles resource allocation to maintain IPC within an acceptable range for serverful workloads. PerfIso [13] isolates CPU interference between latency-sensitive and batch workloads by reserving buffer cores to accommodate load variations in latency-sensitive workloads. However, it still requires one-time profiling for each ongoing foreground task, which may not be feasible in public clouds due to lack of provider control.

SmartHarvest [41] colocates primary customer VMs with VMs running batch jobs. It predicts the peak CPU core usage of primary VMs for each time window using classification models, reallocating idle cores exceeding the peak to batch job VMs. This work targets only CPU usage, and its results are challenging to replicate given that the historical trace and implementation details are not publicly available.

3.2 GPU scheduling for DL workloads

We discuss related works in GPU scheduling under three scenarios: temporal sharing, spatial sharing, and no sharing. The goal of scheduling is often to maximize GPU utilization or reduce total completion time. A common design tradeoff is ensuring job fairness while maintaining high GPU utilization.

Scheduling DL workloads can be approached at three levels of granularity - workload level, iteration level, and kernel level, each suited to different aspects of the workload's execution characteristics.

At the workload level, scheduling is coarse-grained, with decisions made on the order of seconds or minutes, based on the size of workload. This level of scheduling is based on jobs, thus requires the least profiling.

The next level of scheduling is at the iteration level, which ranges from tens of milliseconds to a few seconds. This level is particularly relevant for DL training workloads, where computation is broken down into multiple iterations, each with periodic memory and compute usage patterns. Resource allocations are dynamically adjusted based on the progress of the training or by trimming intermediate results to optimize performance and resource usage.

Kernel-level scheduling offers the finest granularity, operating on the scale of microseconds to milliseconds. This level focuses on the execution of individual GPU kernels, the basic units of computation in GPU workloads. Kernel-level scheduling can significantly impact performance by efficiently managing the execution order and resource allocation of kernels. However, it often incurs significant profiling overheads before execution.

3.2.1 Schedule under temporal sharing

Prior GPU scheduling works involving temporal sharing aim to minimize the cost of state swapping between DL jobs. These works often leverage DL characteristics to reduce context switches and memory oversubscription overheads. However, the primary weakness of temporal sharing is that only one job can be executed at a time, often leading to underutilization of individual GPU kernels.

Antman [45] shares more workloads by swapping cached memory of temporary outputs (data preprocessing, intermediate layer outputs) to host memory at each iteration. To reduce memory copies, the swap only occurs when each mini-batch finishes, as tensors used are identical for each mini-batch. To reduce compute interference, Antman controls kernel launch frequency of opportunistic jobs to ensure SLA requirements of the primary workload. This improves memory and compute utilization by 42% and 35%, respectively, without compromising fairness. However, this approach heavily relies on mini-batch characteristics for scheduling, which is not applicable for inference.

Gandiva [44] introduces a coarse-grained, iteration level checkpoint-restart mechanism to transfer training states between the GPU and host memory during periods of minimum GPU memory usage to reduce data movement cost. This mechanism allows job migrations between nodes and grow-shrink job workers within 1 second, improving cluster utilization by 26%.

TGS [43] adopts an kernel level adaptive rate control mechanism to manage incoming kernels at the OS level for shared containerized workloads. Similar to Antman, TGS ensures a minimum kernel consumption rate for high-priority workloads while reducing kernel executions of opportunistic workloads when GPU is busy by intercepting the CUDA driver API and unifying memory management. This OS-level solution allows applications to run

without manual changes. However, the overheads of GPU page faults when swapping kernels can be significant, as opportunistic workloads might be swapped out frequently.

Gavel [23] focuses on generalizing workload-level scheduling policies with heterogeneous hardware by expressing them as optimization problems. By transforming throughput on different devices into matrices, Gavel can adopt various scheduling objectives (e.g., FIFO, Shortest Job First, Least-Attained Service) and solve linear programming formulations to determine the allocation of jobs for a time slot. Gavel utilizes existing temporal sharing mechanisms to improve utilization, reducing job completion time by 3.5x. However, Gavel requires fixed batch sizes and hardware usage counts specified by the user, lacking the ability to dynamically scale resources based on cluster usage, which is a common technique in large clusters. These related works are discussed further in Chapter 3.2.3.

Our solution does not focus on temporal sharing of the GPU. Instead, we consider spatial sharing, which allows actual concurrent execution without requiring context switches.

3.2.2 Schedule under spatial sharing

Reef [12] improves DL inference serving by enabling kernel-level preemptions on the GPU based on the idempotency of inferece matrix multiplication. Reef also improves GPU utilization by allowing short opportunistic tasks to run concurrently with latency-sensitive tasks, increasing overall throughput by 7.7x with less than 2% latency overhead compared to the always real-time first strategy. However, Reef only supports AMD GPUs due to the need for GPU runtime open-source support and the Apache TVM compiler [1].

Salus [48], like Antman, shares memory space at the iteration level for training tasks by quickly releasing intermediate outputs for shared jobs while persisting model parameters within GPU memory. This is achieved by allocating customized GPU streams and partitioning GPU memory space. Salus eliminates context switches and improves utilization, achieving 7x faster completion time compared to MPS. However, this method is not applicable to inference workloads, which do not include iteration period patterns.

Orion [36] reduces GPU interference via kernel-level scheduling by colocating memory and compute-intensive kernels. To classify whether a kernel is compute-bound or memory-bound, Orion measures the ratio of compute work to data movement offline, and only allows kernels with opposite profiles to colocate. To address the lack of kernel preemption, high and low priority GPU streams are created to facilitate job priority scheduling, which also throttles best-effort jobs if the primary job takes most resources. This approach increases per-GPU request throughput by 7.3x with training job colocations.

However, kernel-level scheduling requires advanced CUDA support. For instance, the compute and memory-bound labeling requires later versions of `NVIDIA_NSIGHT_COMPUTE`

for its roofline analysis function. Additionally, specific CUDA support is needed to set up GPU priority streams for effective scheduling. These requirements limit Orion's adaptability across different GPU product iterations.

Most spatial sharing works require specific hardware support. In contrast, our solution focuses on the workload level, tackling GPU interference from the application perspective with easily accessible kernel metrics.

3.2.3 Schedule without sharing

Some works do not share workloads within a single GPU but focus on scheduling in large clusters where each workload is allocated to one or multiple GPUs. These works prioritize workload speedups over utilization and often require large GPU clusters for scaled evaluations. Common challenges include utilizing heterogeneous GPUs, dynamic resource scaling, and maintaining training efficiency at scale.

Allox [18] observes that small DL workloads have acceptable throughput when running on CPUs. Therefore, it optimizes DL workload scheduling on both GPUs and CPUs. It transforms multi-configuration job scheduling into a min-cost bipartite matching problem by converting time costs into edges. Allox reduces average completion time by up to 95% when system load is high. However, it does not consider GPU sharing, and large DL workloads today are not suitable for CPUs due to higher demands for matrix computations.

Tiresias[10] optimizes scheduling for distributed deep learning (DL) workloads by prioritizing jobs with the highest probability of completing within the next service quantum. To obtain this, distribution of job durations need to be obtained during profiling runs. Also, to enhance decision-making for DL workload placement, Tiresias places models with more skewed tensor distributions within a single node to avoid network congestion costs. As a result, Tiresias improves the average job completion time by up to 5.5 times compared to baseline methods. However, this approach requires profiling to understand tensor overheads and does not account for dynamic resource scaling.

Pollux [33] observes that increasing batch size enhances overall system throughput but decreases training efficiency (training steps required to reach a loss threshold). Based on this observation, Pollux introduces a new scheduling metric, goodput, which trades off training efficiency during low cluster resource pressure to increase overall throughput by dynamically scaling batch size and GPU count. This approach reduces average training time by 37-50%.

Sia [15] builds on Pollux by integrating heterogeneous GPUs with dynamic resource scaling. It creates a goodput estimator that minimizes the goodput search space by profiling job goodput with a few batch sizes on one GPU per type and estimating other configurations by solving an integer linear problem. Sia also introduces a restart penalty score to prevent

frequent checkpointing and restarts on larger models. Sia reduces completion time by 30-93%. Both Pollux and Sia target DL training workloads due to their predictable batch processing patterns.

Chapter 4

System Design

This chapter describes the system design of our design, KACE. The key components of KACE are: (1) Workload profiler, (2) Model trainer, and (3) Colocation predictor; these are depicted in Figure 4.1 along with the interactions between components. The Workload profiler and Model trainer components are offline, while the Colocation predictor works online to determine the workload pair to be colocated.

4.1 Workload profiler

The profiler collects GPU system and kernel metrics for individual workloads offline. These metrics are collected one-time per workload. The metrics are introduced in Chapter 2.3.

Kernel metrics, as mentioned in Chapter 2.3.2, provide detailed insights into individual kernels. An additional advantage of kernel metrics is their relevance to the specific sharing framework in use.

Despite the crucial information (*actual* resource utilization) that kernel metrics provide for interference prediction, profiling kernel metrics can take substantial time [36]. However, the profiling in KACE is done offline, and is thus not on the critical path when selecting the colocation candidate at runtime (see Chapter 4.3). 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 4.1).

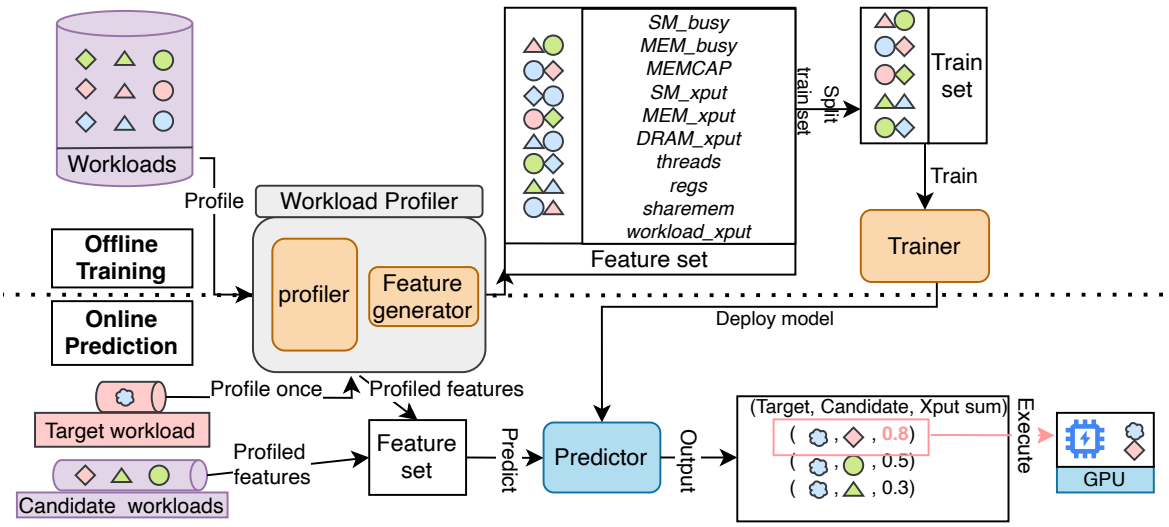

Fig. 4.1 KACE system design showing our feature set.

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.

4.2 Model trainer

To train (and test) a model for predicting the performance of colocated workloads, we run experiments where pairs of workloads are colocated. We believe the system design and methodology of KACE can be extended beyond two colocated workloads, including for non-DL workloads, since the key hurdles we have to overcome to improve colocation performance are not specific to two workloads 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 and monitoring 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 Chapter 6.2.

A key contribution of KACE is its ability to train with a small number of samples. 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 Chapter 6.1; we find that KACE can predict adequately using only 20%–30% of the data for training.

4.3 Colocation predictor

The predictor is an online component that 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 Chapter 5.2 for details. Given a *target* workload to be executed, the predictor iterates through all colocation *candidate* workloads (e.g., from a ready queue [45]) 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 4.1).

4.4 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 [27]. For offline kernel profiling, we use `NsightCompute` [29]; `NVIDIA_MPS` is disabled in this stage. During online prediction runs, we initiate the MPS daemon. Once the predictor determines the ideal candidate workload for colocation, it submits them to the running MPS servers for colocated execution.

Chapter 5

Methodology

This chapter details our experimental setup and evaluation methodology, including the DL workloads employed and the comparison baseline policies for evaluation.

5.1 Evaluation setup

We conduct our evaluation on a single node 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.

5.2 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 [36]. 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).

$$\text{Normalized throughput sum} = \frac{X_1 + X_2}{X_{\text{Oracle}}} \tag{5.1}$$

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 over the 5 runs.

Workload	Batch	SM busy	MEM busy	MEMCAP
BERT-train [5]	2,8,16	97.0%	45.2%	5.1GB
ViT-train [6]	2,8,16	97.2%	37%	17.6GB
ALBERT-train [17]	2,8,16	97.2%	45.1%	7.1GB
BERT-inf [5]	2,8,16	95.1%	38.6%	1.4GB
ViT-inf [6]	2,8,16	28.5%	5.4%	3.2GB
Whisper-inf [34]	2,8,16	44.2%	19.6%	11.7GB
Wav2Vec2-inf [2]	2,8,16	18.9%	6.6%	12.2GB

Table 5.1 Workloads employed in our evaluation. Metrics reported in last three columns are averages across batch sizes.

5.3 Workloads

We employ 21 workloads, encompassing 7 training and inference models with 3 batch sizes each. The workloads have relatively high GPU utilization, representing current GPU models, with an average SM busy rate of over 60%. Detailed workload information is shown in Table 5.1. By considering all possible pairs of workloads that can be accommodated in GPU memory, we obtain 181 possible colocations.

5.4 Comparison baselines

We compare KACE with several baselines, including some 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. [46]* 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 (GPU SM utilization, memory utilization, PCIe read/write bandwidth, VCPU usage, VM memory usage, average number of threads, average kernel length, and long/short kernel ratio) and training 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` [27].
- *Best rule based* is an impractical policy that 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.

Chapter 6

Evaluation

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

6.1 Colocation performance prediction analysis

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 $128\times64\times32$ Deep Neural Network (*NN*), H2O Automatic Machine Learning (*AutoML*) [11], 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 6.1 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 MSE 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 [38]. Additionally, given the relatively small training

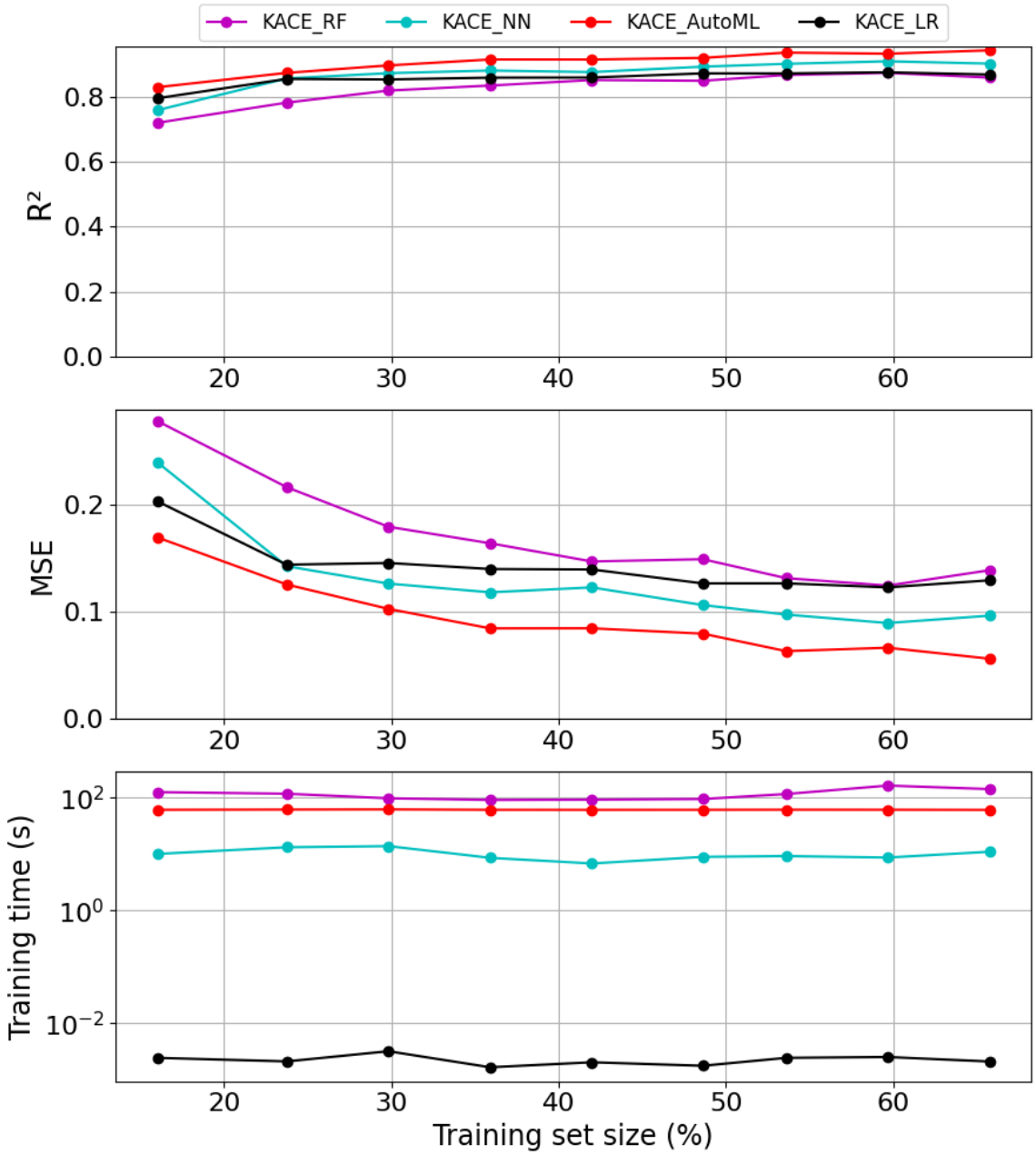

Fig. 6.1 Prediction performance of KACE with different ML techniques as a function of the training data set size.

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 6.1 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

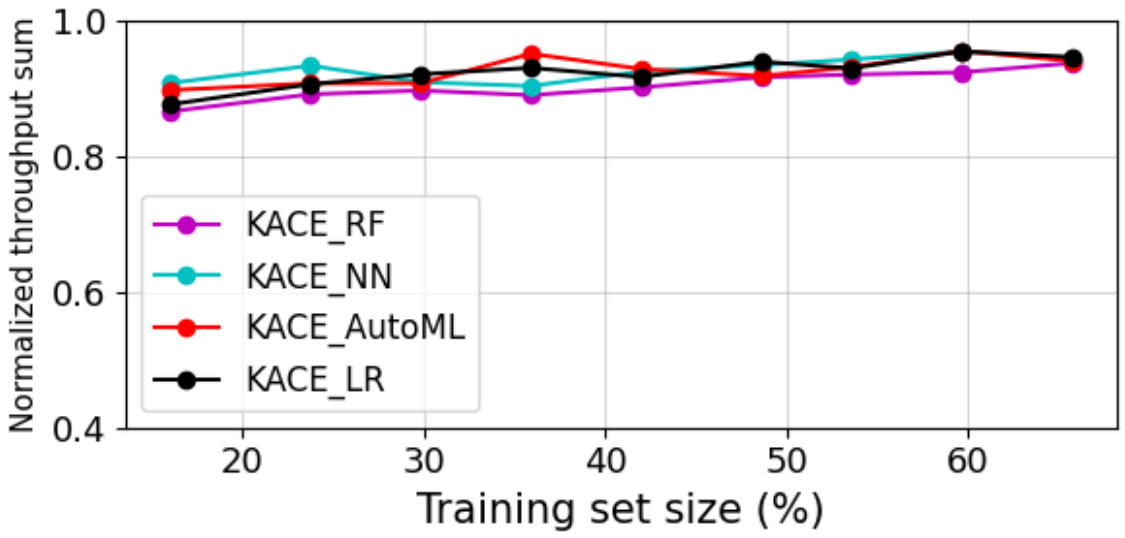

Fig. 6.2 Normalized throughput sum achieved by KACE under different ML techniques.

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. Since throughput can be arbitrarily high, this extrapolation capability allows LR to predict more extreme throughput values (beyond those seen in training) effectively. 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).

6.2 Workload colocation results

We now present our workload colocation results for KACE. We start by evaluating the performance improvement afforded by different ML techniques under KACE.

6.2.1 KACE under different ML techniques

Figure 6.2 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 6.1. 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 for further evaluations.

6.2.2 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 $\sim$20% of the entire dataset (corresponding to about 40 training samples). Results did not qualitatively change with larger training data set sizes.

Figure 6.3 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 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.

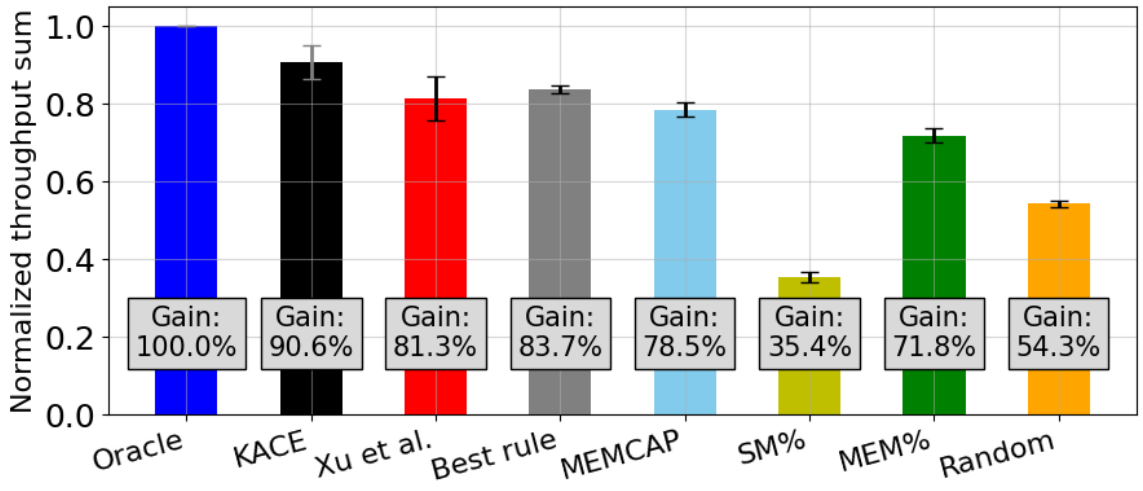

Fig. 6.3 Normalized throughput sum achieved by different colocation policies when using a small training data set size.

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. [46] 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 aggregate kernel metrics, such 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.

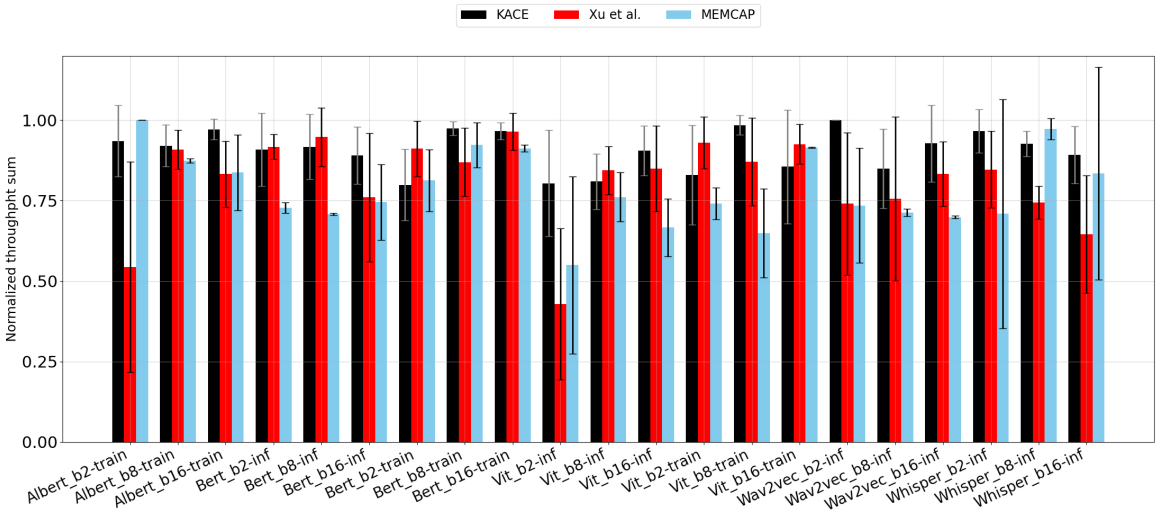

Fig. 6.4 Normalized throughput sum of KACE, Xu et al., and MEMCAP (the best single rule-based policy) for all 21 test cases.

6.2.3 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 6.4; 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

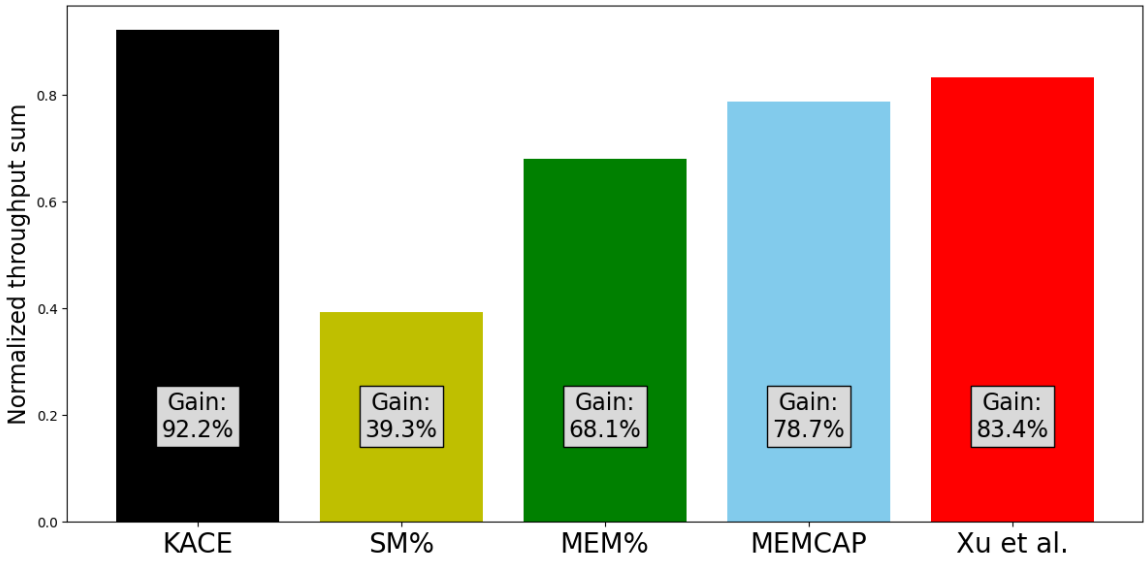

Fig. 6.5 Normalized throughput sum achieved by different colocation policies when the target workload is unseen from the training set.

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 in such scenarios. A similar behavior is also observed when colocating with the Whisper inference workload.

6.2.4 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 with all 7 distinct workloads with the highest batch size, ensuring that the target workload, for any batch size, is not included in training. The results are shown in Figure 6.5. We find that KACE continues to perform well, achieving close to 92% of the throughput sum achieved by Oracle. Compared to Xu et al. and rule-based policies, KACE achieves an 11% higher and a 16% higher average throughput sum, respectively. The above results show that ***KACE performs well even for unseen workloads***.

Chapter 7

Conclusion

In this RPE, we discuss the challenge and opportunities in predicting colocations under GPU spatial sharing. Based on the observations, we 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.

7.1 Future Work

While we have promising results as shown in the previous chapters, we aim to continue working on the following to improve generalizability and expand our work into a real online scheduler.

7.1.1 Short-term future work

- *Extend from colocated pairs to multiple colocated workloads.* In the current preliminary work, we only evaluate interference between two workloads. However, a real-world solution should accommodate more workloads until resources are exhausted or certain SLAs are met. For instance, we should keep adding workloads until the GPU is fully utilized or the throughput sum reaches an acceptable threshold.

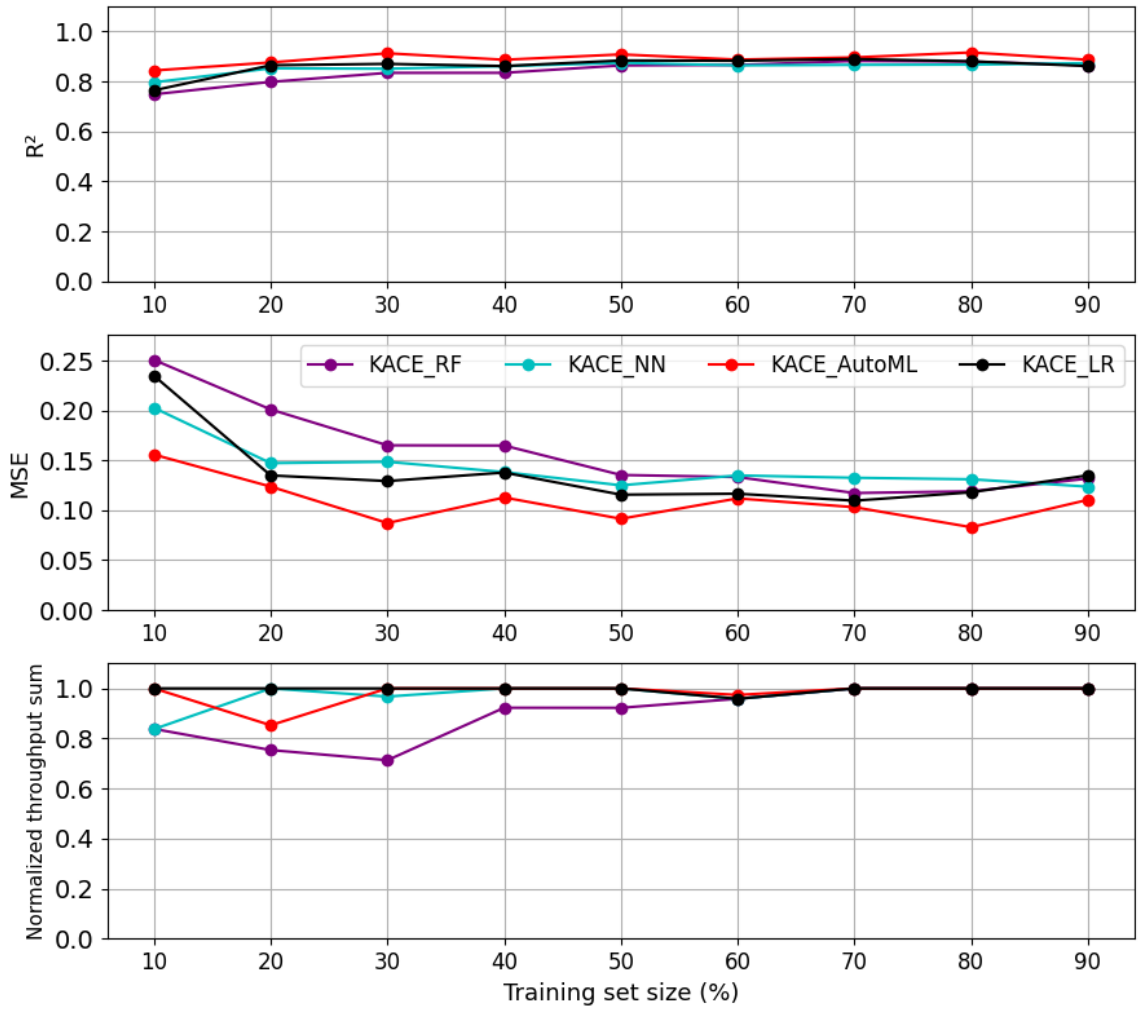

Fig. 7.1 KACE prediction performance of 3-workload colocation

In addition, allowing more than two colocated pairs also generalizes solutions for colocating high-priority and best-effort workloads. This setup maximize GPU usage while maintaining real-time services. For instance, if a lightweight, latency-sensitive inference workload is colocated with multiple other workloads, the system should maintain low latency on the primary inference workload. This is a common use case and should be adopted in our solution.

To demonstrate the extensibility of KACE to multiple workloads, we conducted a trial evaluation to predict the maximum throughput sum of three colocated workloads. For simplicity, we temporarily ignored SLAs and directly evaluated the R^2 and MSE prediction scores, similar to the approach in Figure 6.1. The prediction accuracy and results are displayed in Figure 7.1. The dataset includes 560 colocations with

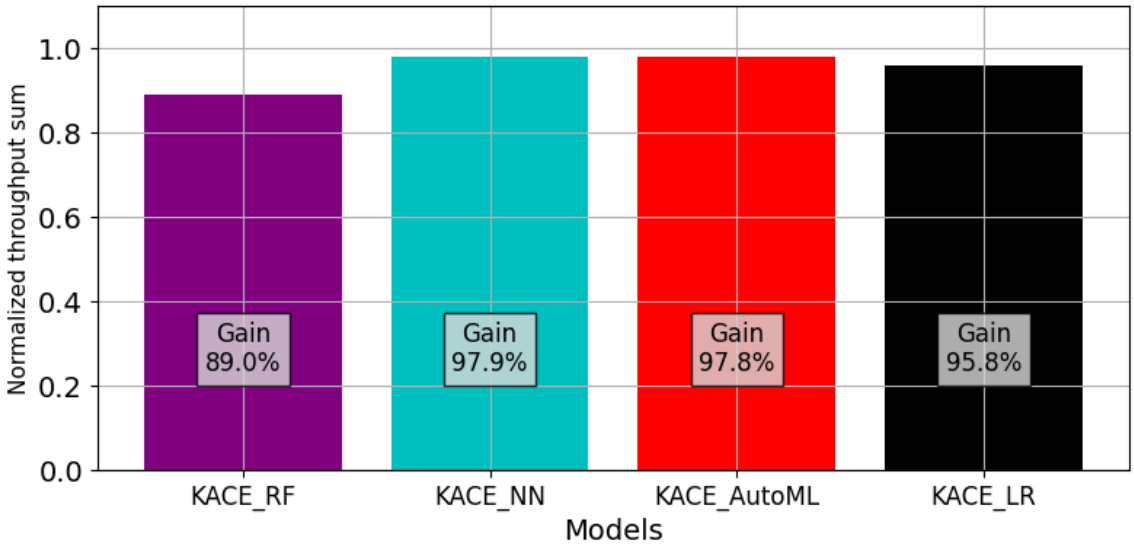

Fig. 7.2 KACE unseen of 3-workload colocation

all 7 workload types, using batch sizes of 2 and 8 in our evaluation. Similar to the 2-workload results, KACE with all ML approaches maintains good R^2 and MSE scores. However, the complexity of the problem necessitates more training data. For instance, with a 10% training set size, 56 colocations are required for 3-workloads, resulting in an MSE value greater than 0.2. In contrast, 2-workload colocation achieves an MSE of 0.2 with 29 workload pairs. With more training data, MSE falls into the low range around 0.15.

The bottom graph in Figure 7.1 shows the predicted normalized throughput sum by selecting the max throughput sum out of the corresponding testing set (size of 560 minus the training set size). We can see that LR and AutoML predict the oracle answer in most cases, while RF has worse results with a small training set. The results indicate that LR performs consistently with multiple workloads.

We also test on 3-workload colocation, with the target workload unseen in the training set. Results are shown in Figure 7.2. The setup is the same as the approach in Chapter 6.2.4, with the normalized throughput sum averaged through all 14 workloads. All models performed well, achieving relative gains over 96% for LR, which is better than the 92% gain for 2-workload colocations. This improvement is due to the availability of more high-throughput candidates to colocate with the target workload, resulting in a higher overall throughput sum. These results also indicate the need to consider other metrics, such as latency and completion time, for a well-rounded evaluation.

- *Enable dynamic thread allocations with MPS.* A key advantage of MPS is its ability to allow flexible thread allocation for each workload by setting `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE`. This functionality provides the opportunity to further optimize workload placements, especially with regard to priorities. For instance, we can accelerate high-priority workloads by assigning 90% of threads to them, while other workloads share the remaining resources. This flexibility is particularly beneficial for colocating multiple lightweight inference workloads with loose time constraints, as they do not require significant GPU compute resources and checkpoint-restarts can be more overhead than slowdowns.

  Another aspect to explore in MPS thread allocation is to model thread oversubscriptions. Thread oversubscription means that the total assigned thread percentage of workloads are over 100%, thus compute resources are shared partially. This is also discussed in related works. For example, one configuration in MISO [19] assigns 50% of threads to all seven colocated workloads, while assigning 14% to each workload achieves exclusive computation. This setup can be beneficial for light-compute jobs, as they have minimal impact on existing jobs. Placing these jobs with exclusive allocation harms overall throughput since other resource-intensive workloads cannot access these resources. By using thread configuration as a control knob, we can model oversubscriptions to further support fairness and priority scheduling.

7.1.2 Long-term future work

- *Interference predictions with DL parallelism.* Our current solution assumes all workloads can be accommodated on a single GPU. However, LLM applications often require multiple GPUs, introducing a new dimension for distributed training and inference. This distributed setup is known as parallelism, and there are several commonly used parallelism paradigms. From a performance analysis perspective, a common bottleneck for all approaches is network efficiency - *how to synchronize intermediate compute results efficiently across machines?* Additionally, with the introduction of faster interconnect protocols like NVLink and InfiniBand, how to fully utilize the terabyte-level bandwidth? These new problems make I/O interference from the network a significant concern, adding a different dimension to compute and memory interference. We aim to explore additional system and kernel metrics to evaluate the interference introduced in this new AI era.

  From a GPU sharing perspective, a parallelized LLM can have fragmented pieces that do not occupy the entire GPU. This introduces a new design problem: *how and when*

should we share fragmented DL models on a GPU? For instance, if two LLMs have small fractions left, should we combine the fractions within one GPU, or would it be better to place them separately, considering the compute and network thresholds of a GPU can be met with a single model? These are open questions to be explored.

- *Scheduling policy for data-intensive GPU applications.* Aside from DL workloads, GPUs are broadly used in interdisciplinary fields such as protein simulations, graphic rendering, and data manipulation. DL workloads are generally more predictable due to similar resource access patterns, while these applications have diverse characteristics. Cloud service providers serve clients with different purposes, posing the challenge of optimizing scheduling among heterogeneous workloads. A naive solution would categorize applications as memory-intensive or compute-intensive and colocate alternative types together. However, detailed kernel metrics might reveal interesting insights to facilitate efficient scheduling in this setup.

References

[1] APACHE SOFTWARE FOUNDATION. Apache tvm: Open deep learning compiler stack, 2023. Accessed: 2023-07-13.

[2] BAEVSKI, A., ZHOU, H., MOHAMED, A., AND AULI, M. wav2vec 2.0: A framework for self-supervised learning of speech representations, 2020.

[3] BARREÑADA, L., DHIMAN, P., TIMMERMAN, D., BOULESTEIX, A.-L., AND CALSTER, B. V. Understanding random forests and overfitting: a visualization and simulation study, 2024.

[4] CORPORATION, N. *CUDA Multi-Process Service Overview*, 2021.

[5] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[6] DOSOVITSKIY, A., BEYER, L., KOLESNIKOV, A., WEISSENBORN, D., ZHAI, X., UNTERTHINER, T., DEHGHANI, M., MINDERER, M., HEIGOLD, G., GELLY, S., USZKOREIT, J., AND HOULSBY, N. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

[7] ELECTRONICS HUB. What is shared gpu memory?, 2023. Accessed: 2024-07-13.

[8] FACE, H. The llama 3 models were trained on 15 trillion tokens with 24,000 gpus, 2023. Accessed: 2024-07-03.

[9] GITHUB. GitHub Copilot. https://copilot.github.com/.

[10] GU, J., CHOWDHURY, M., SHIN, K. G., ZHU, Y., JEON, M., QIAN, J., LIU, H., AND GUO, C. Tiresias: A GPU cluster manager for distributed deep learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (Boston, MA, Feb. 2019), USENIX Association, pp. 485–500.

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

[12] HAN, M., ZHANG, H., CHEN, R., AND CHEN, H. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 539–558.

[13] IORGULESCU, C., AZIMI, R., KWON, Y., ELNIKETY, S., SYAMALA, M., NARASAYYA, V., HERODOTOU, H., TOMITA, P., CHEN, A., ZHANG, J., AND WANG, J. PerfIso: Performance isolation for commercial Latency-Sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 519–532.

[14] JAVADI, A., SURESH, A., WAJAHAT, M., AND GANDHI, A. Scavenger: A black-box batch workload resource manager for improving utilization in cloud environments. In *Proceedings of the 10th ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA, 2019), SOCC '19.

[15] JAYARAM SUBRAMANYA, S., ARFEEN, D., LIN, S., QIAO, A., JIA, Z., AND GANGER, G. R. Sia: Heterogeneity-aware, goodput-optimized ml-cluster scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 642–657.

[16] KEAHEY, K., ANDERSON, J., ZHEN, Z., RITEAU, P., RUTH, P., STANZIONE, D., CEVIK, M., COLLERAN, J., GUNAWI, H. S., HAMMOCK, C., MAMBRETTI, J., BARNES, A., HALBACH, F., ROCHA, A., AND STUBBS, J. Lessons learned from the chameleon testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[17] LAN, Z., CHEN, M., GOODMAN, S., GIMPEL, K., SHARMA, P., AND SORICUT, R. Albert: A lite bert for self-supervised learning of language representations, 2020.

[18] LE, T. N., SUN, X., CHOWDHURY, M., AND LIU, Z. Allox: Allocation across computing resources for hybrid cpu/gpu clusters. *SIGMETRICS Perform. Eval. Rev. 46*, 2 (jan 2019), 87–88.

[19] LI, B., PATEL, T., SAMSI, S., GADEPALLY, V., AND TIWARI, D. MISO: exploiting multi-instance GPU capability on multi-tenant GPU clusters. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, CA, USA, 2022), SoCC '22, p. 173–189.

[20] LI, N., SHANKAR, K., TANG, Z., AND LAHIRI, A. Artificial intelligence for radiographic covid-19 detection: A systematic review. *Data Science and Management 2*, 6 (2021), 100218.

[21] LIM, G., AHN, J., XIAO, W., KWON, Y., AND JEON, M. Zico: Efficient GPU memory sharing for concurrent DNN training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 161–175.

[22] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. 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* (2011), ACM, pp. 248–259.

[23] NARAYANAN, D., SANTHANAM, K., KAZHAMIAKA, F., PHANISHAYEE, A., AND ZAHARIA, M. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 481–498.

[24] NVIDIA. Programming guidelines for gpu architecture, 2013. Accessed: 2024-07-13.

[25] NVIDIA. Nvidia ai enterprise: Deployment guide on vmware, 2021. Accessed: 2024-07-03.

[26] NVIDIA. Achieved occupancy, 2023. Accessed: 2024-07-13.

[27] NVIDIA CORPORATION. *NVIDIA System Management Interface*, Aug. 2016. Version 367.38.

[28] NVIDIA CORPORATION. *NVIDIA Multi-Instance GPU User Guide*, 2021. Accessed: 2024-07-03.

[29] NVIDIA CORPORATION. *NVIDIA Nsight Compute*, 2023. https://developer.nvidia.com/nsight-compute.

[30] OPENAI. ChatGPT. https://openai.com/chatgpt/.

[31] OVERFLOW, S. Interpreting compute workload analysis in nsight compute, 2020. Accessed: 2024-07-13.

[32] OVERFLOW, S. Terminology used in nsight compute, 2020. Accessed: 2024-07-13.

[33] QIAO, A., CHOE, S. K., SUBRAMANYA, S. J., NEISWANGER, W., HO, Q., ZHANG, H., GANGER, G. R., AND XING, E. P. Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)* (July 2021), USENIX Association, pp. 1–18.

[34] RADFORD, A., KIM, J. W., XU, T., BROCKMAN, G., MCLEAVEY, C., AND SUTSKEVER, I. Robust speech recognition via large-scale weak supervision, 2022.

[35] ROBROEK, T., YOUSEFZADEH-ASL-MIANDOAB, E., AND TÖZÜN, P. An analysis of collocation on gpus for deep learning training, 2023.

[36] STRATI, F., MA, X., AND KLIMOVIC, A. Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece, 2024), EuroSys '24, p. 1075–1092.

[37] SURESH, A., AND GANDHI, A. Servermore: Opportunistic execution of serverless functions in the cloud. In *Proceedings of the 2021 ACM Symposium on Cloud Computing* (Seattle, WA, USA, 2021), SoCC '21, pp. 570–584.

[38] TANG, C., GARREAU, D., AND VON LUXBURG, U. When do random forests fail? In *Advances in Neural Information Processing Systems* (2018), S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., vol. 31, Curran Associates, Inc.

[39] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, L., AND POLOSUKHIN, I. Attention is all you need, 2023.

[40] WANG, G., WANG, K., JIANG, K., LI, X., AND STOICA, I. Wavelet: Efficient dnn training with tick-tock scheduling. In *Proceedings of Machine Learning and Systems* (2021), A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, pp. 696–710.

[41] WANG, Y., ARYA, K., KOGIAS, M., VANGA, M., BHANDARI, A., YADWADKAR, N. J., SEN, S., ELNIKETY, S., KOZYRAKIS, C., AND BIANCHINI, R. Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems* (New York, NY, USA, 2021), EuroSys '21, Association for Computing Machinery, p. 1–16.

[42] WIKIPEDIA CONTRIBUTORS. Cuda, 2024. Accessed: 2024-07-13.

[43] WU, B., ZHANG, Z., BAI, Z., LIU, X., AND JIN, X. Transparent GPU sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 69–85.

[44] XIAO, W., BHARDWAJ, R., RAMJEE, R., SIVATHANU, M., KWATRA, N., HAN, Z., PATEL, P., PENG, X., ZHAO, H., ZHANG, Q., YANG, F., AND ZHOU, L. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 595–610.

[45] XIAO, W., REN, S., LI, Y., ZHANG, Y., HOU, P., LI, Z., FENG, Y., LIN, W., AND JIA, Y. AntMan: Dynamic scaling on GPU clusters for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 533–548.

[46] XU, X., ZHANG, N., CUI, M., HE, M., AND SURANA, R. 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, 2019).

[47] YEUNG, G., BOROWIEC, D., YANG, R., FRIDAY, A., HARPER, R., AND GARRAGHAN, P. Horus: Interference-aware and prediction-based scheduling in deep learning systems. *IEEE Transactions on Parallel and Distributed Systems 33*, 1 (2022), 88–100.

[48] YU, P., AND CHOWDHURY, M. Salus: Fine-grained GPU sharing primitives for deep learning applications. *CoRR abs/1902.04610* (2019).