IT UNIVERSITY OF COPENHAGEN

Department of Computer Science

PH.D. DISSERTATION

Collocation and Memory Estimation for Efficient GPU Resource Management in Deep Learning Training

Author:
Ehsan
Yousefzadeh-Asl-Miandoab

Supervisor:
Dr. Pınar Tözün

Last Update
February 14, 2025

Study program, study level:
Computer Science, Ph.D.

Imprint

Project:	Ph.D. Dissertation
Title:	Collocation and Memory Estimation for Efficient GPU Resource Management in Deep Learning Training
Author:	Ehsan Yousefzadeh-Asl-Miandoab
Date:	February 14, 2025
Keywords:	Machine Learning, Deep Learning, GPU, GPGPU, MPS, MIG, Stream, Deep Learning Training, GPU Efficiency, GPU Utilization, Cluster Scheduler, Cluster Resource Manager
Copyright:	IT University of Copenhagen

Study program & University:
Ph.D., Computer Science
IT University of Copenhagen

Supervisor:
Dr. Pınar Tözün
IT University of Copenhagen (ITU)
Email: pito@itu.dk
Web: Link

Abstract

Data science has experienced large-scale and rapid development over the last decade. The main drivers of this development are the availability of a large amount of data, periodically growing computation power, and improving learning and data analysis algorithms. The extensive adoption of deep learning in this field demands computational support for the training phase of the models. For this purpose, enterprises share GPU clusters among different production teams to increase GPU utilization. However, there is sub-optimal utilization of such clusters. This is due to (1) the lack of fine-grained sharing mechanisms of GPUs, (2) scheduling tasks as black boxes, which considers no knowledge about resource requirements of the task.

In this thesis, we start by determining the right set of monitoring tools and metrics that is relevant when reasoning about the hardware utilization of deep learning training. Then, we study collocating deep learning training tasks with the available capabilities of NVIDIA GPUs (GPU streams, MPS, and MIG) to investigate its impact on GPU utilization. While our results emphasize the benefits of collocation, it also demonstrates the challenge of fitting within the available GPU memory as the tasks of different deep learning models collocate. Therefore, as a next step, we propose a machine learning-based mechanism to estimate the GPU memory consumption of deep learning model architectures during training. The estimations are helpful for cluster schedulers and resource managers to map training tasks to processors more efficiently. In the final step, we build a flexible resource manager that provides automated workload collocation allowing different collocation and scheduling options to the end-users. Furthermore, we perform a vast range of experiments and provide the community with our findings and insights.

The findings of this thesis have significant implications for resource-efficient deep learning. As deep learning models continue to grow in size and complexity, efficient GPU utilization becomes a critical challenge. By enabling better workload collocation and accurate GPU memory estimation, this research contributes to reducing wasted computational resources, improving throughput, and making deep learning training more sustainable. These insights are particularly valuable for cloud service providers, research institutions, and enterprises that rely on shared GPU clusters, ensuring that deep learning training workloads run more efficiently, cost-effectively, and with minimal resource wastage.

Resumé

Data science har oplevet en omfattende og hurtig udvikling i løbet af det sidste årti. De primære drivkræfter bag denne udvikling er tilgængeligheden af store mængder data, den kontinuerlige vækst i computerkraft samt forbedringer i lærings- og dataanalysealgoritmer. Den udbredte anvendelse af deep learning inden for dette felt skaber et øget behov for ressourcer, især i træningsfasen af modeller. For at imødekomme dette behov deler virksomheder GPU-klynger mellem forskellige produktionsteams for at maksimere GPU-udnyttelsen. Dog er udnyttelsen af disse klynger ofte suboptimal, primært på grund af (1) manglen på finmaskede delingsmekanismer for GPU'er og (2) planlægning af træningsopgaver som sorte bokse uden kendskab til deres specifikke ressourcekrav.

I denne afhandling identificerer vi først det rette sæt af overvågningsværktøjer og relevante metrikker til at analysere hardwareudnyttelsen i deep learning-træning. Derefter undersøger vi, hvordan deep learning-træningsopgaver kan sameksistere ved at udnytte de eksisterende funktioner i NVIDIA GPU'er, såsom GPU-streams, Multi-Process Service (MPS) og Multi-Instance GPU (MIG), for at vurdere deres effekt på GPU-udnyttelsen. Vores resultater understreger fordelene ved kollokation, men fremhæver også udfordringerne ved at tilpasse sig den tilgængelige GPU-hukommelse, når flere deep learning-modeller trænes samtidigt. Derfor foreslår vi i næste trin en maskinlæringsbaseret metode til at estimere GPU-hukommelsesforbruget af deep learning-modeller under træning. Disse estimater hjælper klyngeplanlæggere og ressourceforvaltere med at tildele træningsopgaver mere effektivt til GPU'er.

Som det sidste trin udvikler vi en fleksibel ressourceforvalter, der muliggør automatiseret arbejdsbelastningskollokation og giver brugerne forskellige kollokations- og planlægningsmuligheder. Derudover udfører vi en bred vifte af eksperimenter og deler vores resultater og indsigter med forskningssamfundet.

Resultaterne af denne afhandling har væsentlige implikationer for ressourceeffektiv deep learning. Efterhånden som deep learning-modeller bliver større og mere komplekse, bliver effektiv GPU-udnyttelse en kritisk udfordring. Ved at muliggøre bedre arbejdsbelastningskollokation og præcise GPU-hukommelsesestimater bidrager denne forskning til at reducere spild af beregningsressourcer, forbedre systemets gennemløb og gøre deep learning-træning mere bæredygtig. Disse indsigter er særligt værdifulde for cloud-tjenesteudbydere, forskningsinstitutioner og virksomheder, der er afhængige af delte GPU-klynger, da de sikrer, at deep learning-træningsopgaver kører mere effektivt, omkostningseffektivt og med minimalt ressourceforbrug.

Acknowledgements

I would like to convey my sincere gratitude to my supervisor, Prof. Pınar Tözün, who consistently went beyond her supervisory duties. Her guidance extended past academic advice into genuine mentorship, offering support through both research challenges and professional development. Her willingness to make time for discussions and thoughtful feedback has been invaluable throughout this journey.

To my parents, I am deeply indebted for your unwavering belief in me. My father's constant encouragement to "try it once more" has been the driving force behind countless successes in my life, while my mother's gentle reminders to maintain work-life balance kept me grounded and healthy throughout this journey. Special recognition to my little brother, Hossein, who has been my emotional anchor and confidant throughout this journey, with Signal app bridging the physical distance between us.

I am especially grateful to Paul, my first friend in Denmark, for the memorable times we shared both as officemates and neighbors. The coincidence of "Paul living in Pulse Living complex near Poul Heningsens Plads metro station" remains an amusing memory.

To my colleagues in 4E10, I truly appreciate sharing the best office in the 4E wing. My heartfelt appreciation goes to all members of DASYA and our meetings, shared treats, and even our bad movies. I extend special recognition to Dovile for her patience in explaining machine learning fundamentals during my Basic and Advanced Machine Learning courses, and to Ties, my academic sibling, with whom I explored the resource-efficient deep learning training landscape for almost two years.

I acknowledge the PhD support team (Jeppe, Julie, Vibe, Sisse, and Espen) and cherish my experience serving as part of PhD Council and Executive Committee for a semester.

I am profoundly grateful to the IT University of Copenhagen for providing an exceptional academic environment. The efficiency of ITU's administrative processes has been remarkable – from day-to-day operations to complex procedures, everything flows smoothly, making it possible to focus entirely on research and academic growth.

I am grateful to Prof. Florina Ciorba for hosting me at the University of Basel for three months. Our intensive four-hour meetings were both challenging and enlightening, helping shape my research perspective. During this visit, I had the pleasure of meeting Shiva, who became not just a colleague but a good friend. What started as a simple idea during our discussions eventually evolved into a research paper

submission.

I extend my sincere recognition to Reza, whose collaboration helped overcome the GPU memory estimation challenges in my research. Our brainstorming sessions, philosophical discussions, and shared commitment to maintaining a healthy work-life balance have been invaluable. Though my thesis writing interrupted our Sunday training sessions, I look forward to resuming them soon.

To Helle and Johnny, I deeply value the warmth and kindness that made Denmark feel like home. Helle and Johnny, our time shared over coffee or meals will always remain special. Johnny, your handyman wisdom has been invaluable. To Hesham, our cooking sessions and the Libyan cuisine adventures, especially "Sidi Mansour," will always bring a smile to my face.

For keeping my spirits high, I acknowledge PureGym, Age of Empires 3, Lego, Garmin, Rammstein, Apache 207, Argy, and Javad Yasari. For providing solace during challenging times, I appreciate Serdar Ortac, Müslüm, Orhan, Ferdi Baba, Mohsen Chavoshi, and Ahmad Azad.

Finally, I express my profound gratitude to Denmark for the opportunities and experiences that have shaped both my academic and personal growth.

List of Figures

List of Tables

Contents

List of Abbreviations

AI	**A**rtificial **I**ntelligence
API	**A**pplication **P**rogramming **I**nterface
CPU	**C**entral **P**rocessing **U**nit
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
DLTTs	**D**eep **L**earning **T**raining **T**ask**s**
CNN	**C**onvolutional **N**eural **N**etwork
DDNN	**D**istributed **D**eep **N**eural **N**etwork
DNN	**D**eep **N**eural **N**etwork
DL	**D**eep **L**earning
DLT	**D**eep **L**earning **T**raining
DRAM	**D**ynamic **R**andom **A**ccess **M**emory
FC	**F**ully **C**onnected
FIFO	**F**irst-**I**n **F**irst-**O**ut
FP	**F**loating **P**oint
GANs	**G**enerative **A**dversarial **N**etworks
GNN	**G**raph **N**eural **N**etwork
GPU	**G**raphical **P**rocessing **U**nit
GPUMemNet	**GPU** **Mem**ory Estimation **Net**work
HBM	**H**igh **B**andwidth **M**emory
HPC	**H**igh **P**erformance **C**omputing
JAX	**J**ust **A**fter e**X**ecution
JCT	**J**ob **C**ompletion **T**ime
LAS	**L**east **A**ttained **S**ervice
LLM	**L**arge **L**anguage **M**odel
LUG	**L**east **U**tilized **G**PU
MAE	**M**ean **A**bsolute **E**rror
MAGM	**M**ost **A**vailable **G**PU **M**emory
MIG	**M**ulti **I**nstance **G**PU
ML	**M**achine **L**earning
MLP	**M**ulti **L**ayer **P**erceptron
MPS	**M**ulti **P**rocess **S**ervice
MUG	**M**ost **U**tilized **G**PU
nvidia-smi	**NVIDIA** **S**ystem **M**anagement **I**nterface
NVML	**NV**IDIA **M**anagement **L**ibrary
NVTX	**NV**IDIA **T**ools e**X**tensions
NVPTX	**NV**IDIA **P**arallel **T**hread e**X**ecution
OBF	**O**racle **B**est **F**it
OFF	**O**racle **F**irst **F**it
OLUG	**O**racle **L**east **U**tilized **G**PU

OOM	**O**ut **O**f **M**emory
OWF	**O**racle **W**orst **F**it
PCA	**P**rincipal **C**omponent **A**nalysis
PC	**P**rincipal **C**omponent
PTX	**P**arallel **T**hread e**X**ecution
QoS	**Q**uality **o**f **S**ervice
RAD	**R**esource **A**ware **D**ata science
RM	**R**esource **M**anager
RR	**R**ound **R**obin
SASS	**S**treaming **ASS**embly
SGD	**S**tochastic **G**radient **D**escent
SLURM	**S**imple **L**inux **U**tility for **R**esource **M**anagement
SMACT	**S**treaming **M**ultiprocessor **ACT**ivity
SMOCC	**S**treaming **M**ultiprocessor **OCC**upancy
S-SGD	**S**ynchronous **S**tochastic **G**radient **D**escent
TC	**T**ensor **C**ore
t-SNE	**t**-Distributed **S**tochastic **N**eighbor **E**mbedding
YARN	**Y**et **A**nother **R**esource **N**egotiator
ZeRO	**Ze**ro **R**edundancy **O**ptimizer

For humanity, freedom, and love . . .

Chapter 1

Introduction

Machine Learning (ML) has become widespread since it could offer better solutions to problems that were hard to address with traditional programming. It puts forward solutions to problems in health, finance, entertainment, science, and engineering fields. For instance, in the medical field, ML models aid doctors in diagnosing their patients. Deep Learning (DL), as a subbranch of ML, has gotten special attention since the 2012 ImageNet competition [1]. In that contest, the DL model, AlexNet [2], defeated other solutions with a dramatically large difference in accuracy for the image classification task. Nowadays, DL models can be found in the core of various services offered by technology giants like Google and Microsoft to medium- and small-sized companies, like Speechify. Language translation, text-to-voice, and image-based search services are examples encompassing DL models at their core [3, 4, 5, 6]. For instance, Speechify's service is a text-to-voice task backed by DL models.

DL models' computing and memory requirements are higher than traditional ML solutions. Furthermore, the size of their models and the datasets they are trained on has increased drastically to boost the models' accuracy over time [7]. The training process is a highly parallel computation. This means these tasks have many sub-tasks that can be executed independently [8]. The case is comparable to a grape farm composed of many lanes (sub-tasks) and work can be done on each lane independently meaning that each worker can work on one lane at a time (thread). Central Processing Units (CPUs) are like strong workers who work fast. On the other hand, Graphics Processing Units (GPUs) are comparable to teams of weaker workers. Consider the scenario of the aforementioned grape farm and the goal is to harvest. Assume that the farm consists of 10 long lanes that need to be harvested. Note that on each lane only one worker can work at a time. A strong worker (comparable to a CPU) can finish harvesting each lane in 1 day, while a weak worker (comparable to a computing core in GPU) finishes a lane in 3 days. If we hire a strong worker (CPU), who can finish the harvest in 10 days, it will cost us as much as the 10 weaker workers. Hiring a team of 10 weaker workers (GPU) can finish the work in 3 days, since each worker can work on each lane of the farm. When it comes to deep learning training tasks, they have a parallelism comparable to the number of farm lanes that should be harvested. Due to this need of extreme parallelism, GPUs are their primary commodity accelerator processors.

TABLE 1.1: First and Latest NVIDIA GPUs' Specifications

	GPU Clock (MHz)	#SMs	Memory (GB)	Memory Bandwidth (GB/s)	FP32 (GFLOPS)	Thermal Design Power (TDP) (Watt)
G80 (2007) [10]	600	16	1.536	76.8	345.6	171
H200 SXM (2024) [11]	1830	> 132	141	4700	67000	700

From 2007 with the emergence of the first programmable GPU, till the end of 2024, GPUs' computing power increased intensely. Table 1.1 shows some of the specification changes of the first and one of the latest NVIDIA data center GPU. It shows around 190x computing power increase in terms of 32-bit floating-point calculations and around 62x improvements in the memory bandwidth. On the other hand, it should be mentioned that also the tasks' computing and memory requirements have increased over time. New applications e.g., more GPU power demanding video games, rendering applications, and deep learning training tasks emerged. Figure 1.1 shows how much computing capability is in demand for pushing AI forward [9].

1.1 GPU Underutilization Challenge

A study analyzing GPU utilization for deep learning jobs in Microsoft cloud over a two month period shows that GPUs remain dramatically underutilized [12], ∼52% utilization on average. Underutilization of computing resources has multiple negative implications: (1) wasted cost on obtaining the computing resources, (2) wasting energy while they burn electricity but do not do actual computing.

One key reason why GPUs remain underutilized is that not all development and data science teams train super-heavy cutting-edge models on gigantic datasets. They train different-sized models and do transfer learning, or their dataset's size limits their model's complexity and size. For example, CNN-based AlexNet has 60 million parameters; on the other hand, the transformer-based Wu Dao 2.0 model [13] has 1.75 trillion parameters. The former one's execution leaves an H200 GPU intensely underutilized, while the latter would fill an H200 GPU. GPU utilization is tightly related to what degree of inherent parallelism an application has. Let's go back to our previous grape farm example, underutilization is comparable to hiring more workers while not having enough work to do or the unsuitability of the work to be shared with more than a specific number of workers.

Another reason for GPU underutilization is GPUs have traditionally lacked advanced virtual memory and fine-grained resource-sharing mechanisms, leading to underutilization issues. Instead of comprehensive virtual memory systems, GPUs employ simplified resource-sharing methods such as NVIDIA's CUDA Streams and

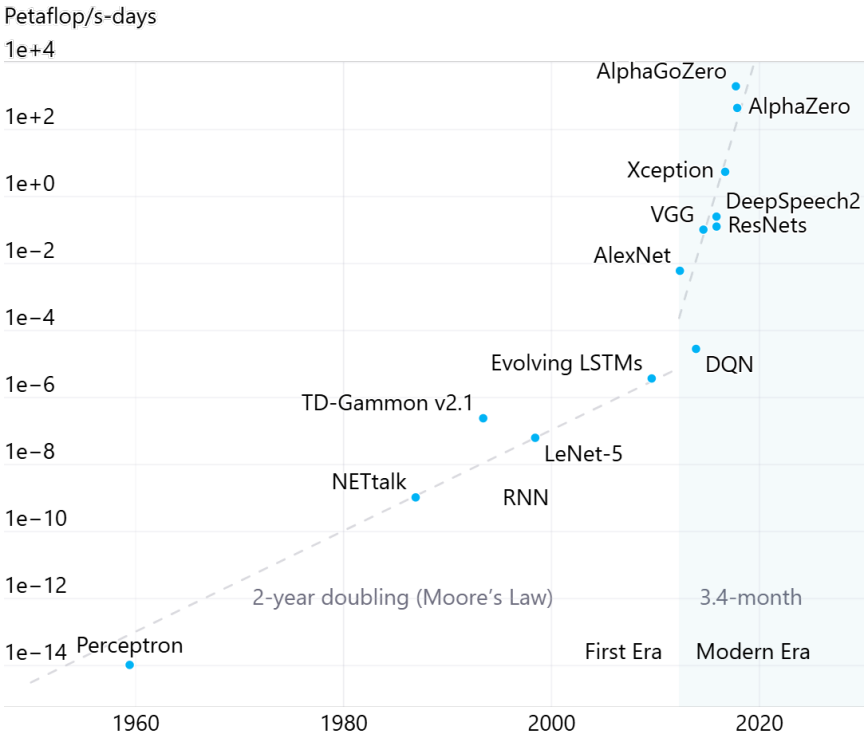


FIGURE 1.1: Training Computation Demand (3.4-month doubling) and Moore's Law Trend (2-year doubling) [9]

Multi-Process Service (MPS). CUDA Streams allow for task serialization, where operations are executed in a sequential manner within each stream. MPS enables spatial resource sharing by allowing multiple processes to share the GPU simultaneously, reducing context-switching overhead. However, MPS does not support memory swapping or paging features, which can result in out-of-memory errors when the combined memory demands of concurrent tasks exceed the GPU's physical memory capacity [14]. This limitation is akin to a team of workers who refuse to take on another job until they have fully completed their current task. In a real-world scenario, such rigid task allocation would be considered a management inefficiency. To illustrate how this contributes to resource underutilization, imagine two neighboring farmers, each with 10 lanes of grape to harvest. They request a company to send a team of workers to assist them. The company sends a team of 30 workers. However, due to an inefficient work policy, the team harvests only one farm at a time, meaning that only 10 workers are actively harvesting while the remaining 20 remain idle and do not work on the other farm. Once the first farm is fully harvested, the team moves to the next farm, again leaving two-thirds of the workforce idle at any given time. Similarly, GPUs, constrained by coarse-grained scheduling and memory limitations, often fail to distribute workloads efficiently, leading to significant resource underutilization.

Finally, on the side of software, adopting HPC/ big data-fit schedulers/ resource managers exacerbates the issue of GPU underutilization. These systems assign GPUs to tasks exclusively and look at the tasks as black boxes [15]. To understand it, consider our grape farm example again, farmers go to a manager who connects them to team workers. This manager assigns one team worker to a farmer ignoring the amount of lanes in that farm and only the team can be hired by a farmer at a time till

the harvest is over. It is problematic as some farms having 10 lanes may end up with a workers team of 200 people and also the farmer is not allowed to share the workers with other farmer(s).

In this thesis, we address the GPU underutilization issue with resource-aware collocation of deep learning training tasks on GPUs at the scheduler and resource manager level and show how much it improves GPU utilization and energy efficiency. Remember the grape farm example, we aim to make the intermediate manager connecting farmers to worker teams to be more resource-intelligent and able to share a team of workers among multiple farms if it is possible. However, making this practical has its challenges. First, we need to know how much computing and memory resources are requried for the deep learning training tasks. It is important if we send two tasks to a GPU and their memory requirements together are higher than what the GPU provides, later coming task will crash because of the out-of-memory error. Another issue can be on-chip resource interference if both of the tasks have a very similar behavior (like both are compute-bound or memory-bound), which causes both tasks to experience longer execution time. At the same time, we need to know the currently available resources for GPUs by monitoring them with representative monitoring metrics. To make it comparable to our farm example, we need to know the number of lanes (representative metric to match with the size of the team we decide to dispatch to do the job), and not only whether the team is actively working in the farm. Furthermore, the number of idle workers of the dispatched team should be monitored.

1.2 Right Tools and Metrics

A fundamental challenge in addressing GPU underutilization is effectively understanding and measuring GPU utilization. Without the right profiling and monitoring toolset and well-defined metrics, inefficiencies in computation become difficult to diagnose, leading to confusion in resource management decisions. Accurate and representative metrics are essential for identifying bottlenecks, as simplistic GPU utilization metrics often fail to reflect how GPU computing resources are actually used [16]. This limitation arises from the common definition of GPU utilization, which considers a GPU as "utilized" whenever at least one thread is executing on it within a given time slot, regardless of overall efficiency.

Additionally, different profiling and monitoring tools have their own advantages, limitations, and computational overheads, making it crucial to evaluate their effectiveness in specific scenarios. Choosing the right combination of metrics and tools is therefore a prerequisite for conducting informed analyses and implementing meaningful optimization efforts across various abstraction layers. In Chapter 3, we provide a detailed evaluation of available profiling and monitoring tools and metrics, identifying the most representative ones for online, real-time decision-making, a critical requirement for schedulers and resource managers.

Understanding GPU utilization requires more than a simplistic utilization metric;

selecting the right profiling and monitoring tools with representative metrics is essential for accurate diagnosis, efficient resource management, and informed optimization strategies across different abstraction layers.

1.3 Understanding Collocation

Task collocation is a fundamental technique for improving GPU utilization by enabling multiple training tasks to run concurrently on the same GPU [17]. One of the key steps in enhancing GPU resource efficiency is understanding how workload collocation affects both performance and energy consumption. While collocation has the potential to increase GPU load and utilization, it also introduces challenges such as resource interference and memory contention, which can lead to unpredictable slowdowns and out-of-memory (OOM) crashes. Without a systematic approach to benchmarking and analyzing collocation, its benefits and trade-offs remain unclear, limiting its practical applicability in the design of efficient schedulers and resource managers.

To address this, Chapter 4 presents a detailed analysis of collocating deep learning training tasks on GPUs. This study evaluates how various collocation strategies impact GPU performance metrics, considering factors such as model architecture, batch size, and memory bandwidth demands. Our results demonstrate that while collocation can significantly enhance GPU utilization, its effectiveness depends on workload characteristics and system-level configurations. Additionally, this work explores the available NVIDIA mechanisms for collocation, providing insights into their benefits and limitations.

Selecting an optimal collocation strategy is therefore essential for developing efficient scheduling and resource management solutions. A well-informed, collocation-aware resource manager can balance workload execution, minimize resource contention, and maximize overall GPU throughput. By systematically benchmarking and analyzing the impact of collocation in deep learning training, this thesis lays the foundation for more intelligent resource management and allocation strategies that leverage GPU resources more effectively.

Collocation can enhance GPU utilization, but its benefits depend on a deep understanding of resource interference, memory contention, and workload interactions. Benchmarking and analyzing collocation is therefore crucial for designing efficient schedulers and resource managers that maximize GPU efficiency while minimizing performance degradation.

1.4 Estimating GPU Memory Requirement of Training Tasks

Understanding and predicting GPU memory requirements for deep learning training tasks is crucial for avoiding out-of-memory (OOM) crashes, particularly in collocated workloads. Deep learning models, including MLPs, CNNs, and Transformers,

exhibit diverse memory usage patterns based on factors such as architecture, batch size, and parameter count. Without an accurate estimation of GPU memory consumption, schedulers and resource managers lack the necessary insights to make informed decisions when mapping tasks to shared GPU resources. This lack of foresight often leads to inefficient resource allocation, unexpected OOM errors, and overall suboptimal utilization of GPU clusters.

To address this, in Chapter 5, we present GPUMemNet, a framework for building a dataset and training predictive models to learn the patterns governing GPU memory usage in deep learning workloads. This study collects extensive profiling data from MLP, CNN, and Transformer architectures, capturing key features that influence memory consumption during training. By leveraging this dataset, we train machine learning models to predict memory requirements based on network characteristics, hyperparameters, and training configurations. The resulting models provide valuable insights that can guide scheduling and resource management policies, enabling safer and more efficient collocation of workloads. Also, we make the dataset we built and trained our models on publicly available.

Integrating GPUMemNet into scheduling frameworks has the potential to reduce OOM crashes by allowing schedulers to anticipate memory demands before assigning tasks to GPUs. A memory-aware scheduler can dynamically allocate resources, prevent memory overcommitment, and ensure stable execution across shared environments. By systematically analyzing and modeling GPU memory usage, this thesis contributes to the development of more intelligent and robust resource management strategies for deep learning workloads.

Predicting GPU memory usage is essential for preventing OOM crashes in collocated workloads. By building GPUMemNet to model memory consumption patterns, this work provides critical insights for schedulers and resource managers, enabling smarter task allocation and more efficient GPU utilization.

1.5 Collocation-Aware Resource Manager

Maximizing GPU utilization and energy efficiency in deep learning training requires a comprehensive approach that integrates accurate profiling, collocation analysis, and memory estimation into resource management decisions. Throughout this thesis, we have identified the right set of profiling tools and metrics, studied the impact of collocation on GPU performance, and developed GPUMemNet, a predictive model for estimating GPU memory requirements. Building upon these foundations, Chapter 6 presents the design, implementation, and evaluation of a collocation-aware resource manager that leverages these insights to enable efficient and intelligent resource management.

The proposed resource manager incorporates collocation as a fundamental scheduling strategy, ensuring that deep learning workloads are assigned to GPUs in a way that maximizes resource utilization while minimizing performance degradation and avoiding OOM crashes. To achieve this, we design and evaluate a set of collocation

policies that balance compute efficiency, memory constraints, and energy consumption. These policies make use of profiling data, memory usage predictions, and workload characteristics to dynamically determine which tasks can be collocated safely and efficiently.

Moreover, despite careful profiling and predictive modeling, OOM crashes may still occur due to unpredictable factors such as GPU memory fragmentation or slight mispredictions in memory estimation. To address this, we develop a simple yet effective recovery mechanism that detects and handles OOM failures, allowing the scheduler to send them again for execution. This mechanism ensure the reliability of the system.

The results demonstrate that strategic collocation decisions can significantly improve GPU utilization while reducing energy waste, offering a promising direction for large-scale deep learning infrastructure. By integrating insights from monitoring the hardware resources, and memory estimation from tasks, this resource manager provides a practical framework for optimizing deep learning training workloads in shared environments.

Collocation-aware scheduling is key to improving GPU utilization and energy efficiency in deep learning training. By combining profiling, memory estimation, intelligent policy design, and an OOM recovery mechanism, this work presents a practical resource manager that enables efficient workload collocation, reduces underutilization, and ensures system stability.

1.6 Thesis Statement and Contributions

This thesis contributes to tackling GPU underutilization challenge and improving performance and energy-efficiency of shared GPU clusters for deep learning training.

Thesis Statement

"To improve GPU utilization and energy efficiency for deep learning training, one must build resource managers capable of intelligent workload collocation on GPUs. Such resource managers will require a memory estimation tool for deep learning workloads to ensure effective collocation of deep learning tasks."

The contributions of this thesis are summarized as follows:

- Comprehensive Analysis of GPU Profiling and Monitoring Tools
  - We conduct an in-depth evaluation of existing GPU profiling and monitoring tools, identifying key metrics that accurately reflect GPU utilization during deep learning training.
  - Our analysis provides insights into the strengths and limitations of various tools, guiding the selection of appropriate methods for real-time

decision-making in resource management or identifying the bottlenecks for optimizations in different levels.

- Systematic Study on Deep Learning Training Task Collocation
  - We investigate the impact of collocating multiple deep learning training tasks on a single GPU, analyzing how different collocation strategies affect performance, memory usage, and energy consumption.
  - Our findings highlight the conditions under which collocation is beneficial and outline potential pitfalls, informing the design of more effective resource management policies.
- Development of **GPUMemNet** for Memory Usage Prediction
  - We review the available predictive methods for GPU memory requirement of deep learning tasks and show their inefficiency and inaccuracy.
  - We introduce GPUMemNet, a dataset with a set of predictive models designed to estimate the GPU memory requirements of various deep learning workloads.
  - By leveraging a comprehensive dataset of neural network training tasks, GPUMemNet enables resource managers to make informed decisions, reducing the risk of out-of-memory errors during task execution.
  - We pave the way for building GPU utilization estimators and encourage the community to develop them by releasing all our artifacts.
- Design and Implementation of a Collocation-Aware Resource Manager
  - Building upon our profiling, collocation analysis, and memory prediction, we develop a resource manager that intelligently schedules deep learning tasks on shared GPU clusters at the server scale.
  - We introduce a recovery mechanism in case of training errors and incorporate it to the resource manager as well.
  - The proposed final system dynamically allocates resources based on real-time monitoring and predictive insights, enhancing GPU utilization and energy efficiency while maintaining system stability.

The aforementioned contributions resulted in the following publications, open-source software, and dataset:

- Publications
  - **Ehsan Yousefzadeh-Asl-Miandoab**, Ties Robroek, and Pınar Tözün. "**Profiling and monitoring deep learning training tasks.**" Proceedings of the 3rd Workshop on Machine Learning and Systems. 2023.

  - Ties Robroek, **Ehsan Yousefzadeh-Asl-Miandoab**, and Pınar Tözün. "An Analysis of Collocation on GPUs for Deep Learning Training." Proceedings of the 4th Workshop on Machine Learning and Systems. 2024.

  - (To be submitted) Chapters 5 and 6 as "**Collocation-aware Resource Management with GPU Memory Usage Estimations**"

- Open-source Tools and Datasets

  - **GPUMemNet Artifacts**
    `https://github.com/ehsanyousefzadehasl/GPUMemNet`
    This repository includes the dataset and all scripts developed throughout the entire process, encompassing data gathering, cleaning, analysis, and training notebooks.

  - **Resource-Aware Data Science Resource Manager (RAD-RM)**
    `https://github.com/ehsanyousefzadehasl/RAD-RM`
    This repository includes the source code for the resource manager, encompassing all models, the Philly trace analyzer, and the workload mapper utilized in evaluating the proposed policies and mechanisms in this thesis.

Beyond the scope of this thesis, I have also contributed to the following publications and open-source software:

- Publications

  - Ties Robroek, Aaron Duane, **Ehsan Yousefzadeh-Asl-Miandoab**, and Pınar Tözün. "**Data Management and Visualization for Benchmarking Deep Learning Training Systems.**" DEEM 2023

  - (Under Review) Shiva Parsarad, **Ehsan Yousefzadeh-Asl-Miandoab**, Florina M.Ciorba, Pınar Tözün, and Isabel Wagner. "**DP-Morph: Improving the Privacy–Utility–Performance Trade-off for Differentially Private OCT Segmentation**"

- Open source tools

  - **radT: Resource Aware Data science Tracker**
    `(https://github.com/Resource-Aware-Data-systems-RAD/radt`
    Data tracking and management tool introduced at DEEM.

1.7 Roadmap

The remainder of this thesis is organized as follows:

- Chapter 2 provides the necessary **background and related work** on tackling GPU underutilization challenge. This chapter, after building the foundation for understanding the concepts of the research, discusses proposed techniques

and methods for resource management in the community, their limitations, and open research questions that motivate the contributions of this thesis.

- Chapter 3 presents a detailed study on **profiling and monitoring tools and GPU utilization metrics**. This chapter explores the right set of metrics and tools required to have a fine-grained understanding of GPU utilization. It evaluates the strengths and limitations of different profiling tools and establishes a methodology for collecting meaningful performance data, which serves as a foundation for later chapters.

- Chapter 4 investigates **collocating deep learning training tasks on NVIDIA GPUs**. It analyzes three different collocation strategies, evaluating their impact on GPU utilization, memory efficiency, and performance stability. The findings from this study provide essential insights into designing better resource management policies.

- Chapter 5 introduces **GPUMemNet**, a framework for building a dataset and training predictive models to estimate GPU memory requirements for deep learning workloads. This chapter details the data collection process, key features extracted, and model training methodologies used to predict GPU memory consumption. The goal of this work is to provide schedulers with accurate memory estimations to prevent out-of-memory (OOM) crashes when collocating tasks.

- Chapter 6 presents the **design, implementation, and evaluation of a collocation-aware resource manager**. This scheduler incorporates the insights gained from profiling and monitoring tools and metrics analysis, collocation analysis, and memory estimation to intelligently allocate tasks to GPUs. It also introduces a simple recovery mechanism to handle unexpected OOM crashes due to factors like memory fragmentation or mispredictions. Furthermore, this chapter explores various collocation policies, evaluating their effectiveness in increasing GPU utilization and energy efficiency.

- Finally, Chapter 7 summarizes the key contributions of this thesis, reflects on its broader implications for GPU resource management in deep learning, and outlines potential directions for **future research**.

Chapter 2

Background and Motivation

Deep learning has revolutionized various fields by enabling data-driven solutions to complex problems, but its success comes with significant computational and resource management challenges. This chapter provides an overview of deep learning and widely used machine learning frameworks that support model development and training. We then discuss the role of GPUs in accelerating deep learning workloads and explore different GPU sharing mechanisms, including multi-process service (MPS) and multi-instance GPU (MIG). Furthermore, we examine existing schedulers and resource management systems, such as SLURM, and their limitations in handling deep learning workloads. Finally, we highlight open challenges in resource management and scheduling for deep learning, setting the stage for our proposed approach in later chapters.

2.1 Deep Learning Training

Unlike classical machine learning algorithms, which often require manual feature engineering, deep learning models learn hierarchical representations directly from raw data. This ability to automatically extract meaningful patterns has enabled breakthroughs in areas such as image classification, speech recognition, natural language processing, and autonomous driving. By leveraging large-scale data and neural networks with many layers, deep learning provides superior accuracy and adaptability, even in scenarios where crafting explicit rules or heuristics is impractical.

However, the success of deep learning depends heavily on the availability of large and diverse datasets, which are often challenging to construct. High-quality datasets for deep learning require extensive data collection, accurate labeling, and preprocessing to ensure consistency. Manual annotation, a common approach for tasks like image segmentation or object detection, is labor-intensive and prone to human error [18]. In some domains, such as healthcare or autonomous systems, data privacy, security, and accessibility further complicate dataset creation. Moreover, the need for domain-specific datasets that are both representative and unbiased is critical for ensuring that deep learning models generalize well to real-world scenarios [19].

Training deep learning models is computationally intensive, often requiring specialized hardware like GPUs to handle the massive matrix operations involved in

forward and backward propagation [20]. The increasing complexity of modern architectures, such as transformers [4] and generative adversarial networks (GANs) [21], demands ever-higher computational resources and memory capacity. Large-scale models may involve billions of parameters, necessitating distributed computing across multiple GPUs or even entire clusters. Memory limitations on GPUs can restrict batch sizes, slowing down training and requiring optimization techniques to fit models into available resources.

Additionally, the energy consumption of deep learning training is a growing concern [22]. Training state-of-the-art models can take days or weeks, consuming significant electricity and contributing to the carbon footprint of AI research. Techniques such as mixed-precision training, model pruning, and transfer learning are being actively developed to mitigate these issues by reducing resource requirements without compromising performance. Despite these challenges, the potential of deep learning to revolutionize industries continues to drive innovation in both algorithms and infrastructure.

2.1.1 Deep Learning Training Process and CPU vs. GPU Execution

The training process of deep learning models consists of multiple stages, each involving different computational tasks. These tasks are executed either on the CPU or GPU, depending on their computational characteristics.

Data Preprocessing (CPU-Intensive)

Before training begins, raw data undergoes several preprocessing steps, such as resizing, normalization, data augmentation (for images), and tokenization (for text). These operations typically run on the CPU because:

- They involve irregular memory access patterns and file I/O.
- They are not highly parallelizable and often require interaction with storage devices.
- Many deep learning frameworks, such as PyTorch and TensorFlow, utilize CPU-based multi-threading for efficient data pipeline management.

To reduce preprocessing overhead, frameworks implement data loaders with prefetching and parallel processing, ensuring a steady supply of batches to the GPU.

Model Initialization and Forward Pass (GPU-Intensive)

Once data is prepared, the neural network processes input data in the forward pass, where:

- Data propagates through layers (e.g., convolutional, recurrent, or transformer blocks).

- Matrix multiplications, convolutions, and activation functions are applied.

Since these computations involve massive tensor operations, GPUs are used due to their high parallelism and optimized linear algebra libraries such as cuDNN and cuBLAS.

Loss Computation and Backpropagation (GPU-Intensive)

During training, the model's predictions are compared with ground-truth labels using a loss function (e.g., cross-entropy, mean squared error). Then, backpropagation computes gradients for each model parameter using the chain rule:

- Gradient computation involves large-scale matrix differentiation.
- GPUs accelerate these calculations through parallel execution.

Backpropagation is the most computationally expensive phase, requiring high memory bandwidth and efficient gradient computations.

Weight Updates and Optimization (GPU/CPU Hybrid)

Once gradients are computed, model parameters are updated using an optimizer (e.g., stochastic gradient descent, Adam). This process can be executed on:

- The **GPU**, if all model parameters fit within GPU memory.
- The **CPU**, in distributed settings where updates are aggregated across multiple GPUs.

Optimized implementations such as mixed-precision training reduce memory usage while maintaining numerical stability.

Checkpointing and Logging (CPU-Intensive)

Periodically, model parameters and training metrics (e.g., loss, accuracy) are saved to disk. This is typically performed on the CPU because:

- File I/O operations are not GPU-accelerated.
- Logging frameworks such as TensorBoard and Weights & Biases manage training metadata asynchronously to avoid bottlenecks.

Inference and Post-Processing (CPU/GPU Hybrid)

Once trained, models can be deployed for inference, with execution depending on the use case:

- **GPU-based inference** is preferred for real-time applications (e.g., autonomous driving, robotics).

- **CPU-based inference** is common in production settings where energy efficiency is crucial.

Post-processing steps such as non-maximum suppression (for object detection) or token decoding (for NLP tasks) are often CPU-bound due to their sequential nature.

2.1.2 Challenges and Optimization Strategies

Despite the advantages of GPU acceleration, training large-scale deep learning models presents challenges:

- **Memory Constraints**: High-dimensional tensors require significant memory. Techniques like mixed-precision training and gradient checkpointing help mitigate GPU memory bottlenecks.
- **Batch Size Trade-offs**: Large batch sizes improve GPU utilization but require more memory, necessitating careful tuning.
- **Energy Consumption**: Training models at scale is energy-intensive. Efficient architectures, model pruning, and hardware-aware training strategies are actively researched to improve sustainability.

Deep learning continues to evolve, with innovations in hardware and algorithmic efficiency driving advancements across industries. Understanding the interplay between CPUs and GPUs in training workflows is essential for optimizing performance and scalability.

2.2 Machine Learning Frameworks

Machine learning frameworks are essential tools that simplify the development, training, and deployment of machine learning models. These frameworks abstract the complexities of mathematical operations and low-level programming, enabling researchers and developers to focus on designing and refining models. By providing features such as automatic differentiation [23], pre-built components, GPU acceleration, and support for distributed training, frameworks significantly accelerate the pace of innovation in machine learning. Among the most popular frameworks are PyTorch [24], TensorFlow [25], Just After eXecution (JAX) [26], and MXNet [27], each of which has distinct advantages tailored to specific research and production needs.

PyTorch

PyTorch [28], developed by Facebook AI Research, is one of the most widely used frameworks in machine learning. It is particularly favored in academia for its dynamic and flexible nature, which makes it ideal for research and experimentation. A key feature of PyTorch is its support for dynamic computation graphs, also known as eager execution, which allows operations to be executed immediately without pre-defining the entire computational structure. This feature makes debugging and prototyping highly intuitive. PyTorch includes an extensive ecosystem of libraries, such as

TorchVision for computer vision tasks, TorchText for natural language processing, and TorchAudio for audio-related applications. It also supports GPU acceleration through CUDA and enables distributed training for large-scale models. These features make PyTorch a versatile choice for a wide range of machine learning tasks, from natural language processing to generative modeling.

TensorFlow and Keras

TensorFlow [25], developed by Google, is a comprehensive machine learning framework designed to cater to both research and production environments. Initially, TensorFlow used static computation graphs, but with the release of TensorFlow 2.x, it adopted eager execution, improving usability for researchers. TensorFlow offers a robust set of tools for production, including TensorFlow Serving for deploying models in production environments and TensorFlow Lite for optimizing models for mobile and embedded devices. Additionally, its high-level Keras API simplifies the process of building and training models, making it accessible for users of all expertise levels. TensorFlow excels in handling large-scale applications, with strong support for distributed training and deployment pipelines. Its versatility and production-ready features make it a preferred choice for industrial applications and large-scale deployments.

Keras [29] is a high-level, user-friendly interface for building and training machine learning models. Originally developed as an independent library, Keras is now tightly integrated with TensorFlow as its official high-level API. Keras acts as a wrapper over complex, low-level libraries like TensorFlow, Theano [30], and CNTK [31], abstracting the complexities of neural network programming and offering a simplified, modular approach to model development. This abstraction makes Keras particularly appealing for beginners and practitioners seeking to rapidly prototype deep learning models. Its core design principles—simplicity, modularity, and extensibility—allow users to build complex models using an intuitive interface with minimal code. Keras provides pre-built layers, optimizers, loss functions, and metrics, enabling users to focus on high-level design without delving into implementation details.

2.3 GPU Computing

GPUs are general-purpose parallel accelerators that have proven their efficiency in a variety of parallel tasks, including graphics processing and, more recently, deep learning training. Initially, GPUs were introduced as co-processors for accelerating graphical applications like video games. At that time, programming GPUs required the use of shader programming languages such as OpenGL and Microsoft DirectX, which demanded deep expertise in computer graphics, GPU processor architecture, data mapping to graphical constructs, and interpreting results from graphical data [32].

In 2007, NVIDIA introduced CUDA, a parallel computing platform and application programming interface (API), which revolutionized GPU programming. CUDA made it significantly easier to program GPU devices for non-graphical applications,

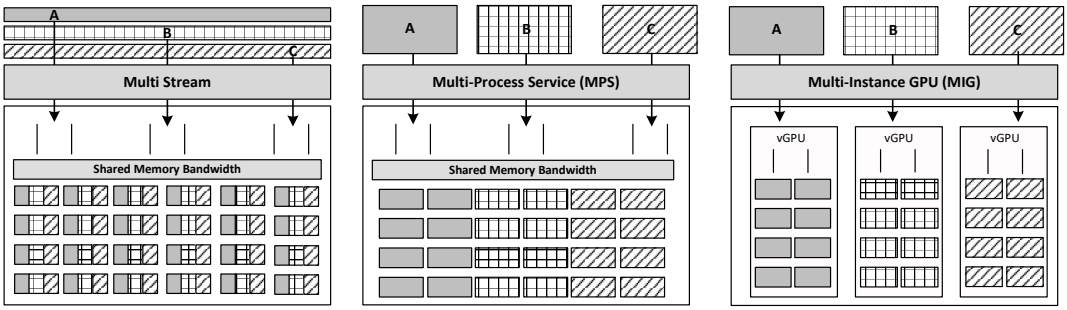


FIGURE 2.1: Three GPU sharing options; A, B, and C represent the applications sharing a GPU

earning them the name General-Purpose GPUs (GPGPUs) [33]. This marked a shift, enabling researchers and developers to leverage GPUs for a broad range of compute-intensive applications.

One of the most transformative applications of GPUs has been in the fields of machine learning and deep learning. These domains rely heavily on large-scale matrix operations, which GPUs, with their thousands of parallel cores, are exceptionally well-suited to handle. Unlike traditional CPUs, which are optimized for sequential processing, GPUs excel at performing many calculations simultaneously, making them ideal for training deep neural networks [34]. The ability to process large datasets and perform complex computations efficiently has made GPUs indispensable in machine learning workflows. This accelerated the development of groundbreaking technologies like image recognition, natural language processing, and generative models, propelling machine learning and deep learning to the forefront of artificial intelligence research and applications.

Modern GPUs have grown exponentially in their computation and memory capabilities and they suffer from underutilization. It means that the GPU consumes energy and wears out but is not fully used. Over-provisioning, which means running more than one application on a GPU, is one of the mainstream methods to tackle the challenge. However, unlike CPUs, GPU architecture lacks sophisticated resource-sharing methods like virtual memory and fine-grain sharing.

There are three approaches to executing several applications on the currently available Nvidia GPUs: submitting more than one application with various streams [35], with multi-process service (MPS) [14], and splitting GPU into smaller isolated GPUs with MIG [36]. Figure 2.1 shows the difference between these options from interference level, which is delved into in the following text.

GPU Streams

CUDA 7 introduced the option of running multiple processes simultaneously using their own CUDA stream on the same GPU. A CUDA stream is a sequence of operations that execute on the GPU (i.e., kernels and data transfers) in the order in which the host code issues them. While operations within a stream are guaranteed to

execute in the prescribed order, operations in different streams can run concurrently. This concurrency (interleaving of the streams) greatly helps with overlapping the data transfers between the host CPU and GPU [35]. From now on, we call this type of workload collocation the **naive** method in this document. In naive collocation, the opportunity for sharing the hardware is limited. This is because the streams have to share the GPU compute resources in a **time-based** manner rather than having resources explicitly dedicated for each stream.

GPU MPS

NVIDIA introduced the **Hyper-Q** technology with Kepler architecture in 2012. Hyper-Q enabled several CPU threads to launch kernels on a single GPU resulting in increased GPU utilization and decreased CPU idle times. Hyper-Q also eliminates **false dependencies** across different applications to increase GPU utilization. Before Hyper-Q, different threads could submit tasks on different streams (CUDA 7+). The work distributor used to take work from the front of the pipeline and assign work on the available SMs after checking all dependencies are satisfied. With Hyper-Q, a grid management unit (GMU) was introduced. GMU creates multiple hardware work queues to reduce or eliminate false dependencies [37]. The *multi-process service (MPS)* utilizes **Hyper-Q capabilities**. Similar to streams, MPS processes share GPU memory and bandwidth. However, unlike streams, the GPU's streaming multiprocessors (SMs) are split across the different processes. This split is done by the MPS daemon automatically based on the provisioning of the GPU compute resources needed for each process. However, CUDA programmers can have control over how resources are split as well. While this provisioning introduces some process management overhead, splitting resources this way offers reduced interference across the different processes. One limitation of MPS is that the processes have to be launched by a single user for security reasons. Therefore, MPS cannot collate applications launched by different user accounts [38, 14].

GPU MIG

Multi-Instance GPU (MIG) is the most recent collocation technology introduced with NVIDIA's Ampere GPUs. It introduces hardware support for splitting a GPU into smaller GPU instances of varying sizes. These GPU instances may run different processes allowing them to run in parallel on the same GPU. On the hardware side, MIG-capable GPUs are divided up into multiple slices. These can be combined into GPU instances, providing a partitioning of the GPU. The memory of the A100 GPU is split into 8 memory slices, and the compute side is split into 7 compute slices, plus one reduced slice for the partition management overhead. A limitation of enabling MIG is that it does not allow for GPU-to-GPU communication in the multi-GPU training cases. On the other hand, each partition is strictly separated regarding hardware resources, preventing interference across partitions. Figure 2.2 shows the possibilities of splitting an A100 GPU [36, 39].

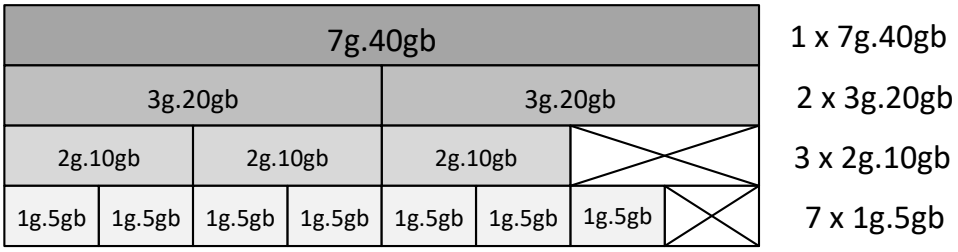


FIGURE 2.2: A100 GPU MIG partitioning possibilities [36, 39]

2.4 Schedulers and Resource Management Systems

Schedulers and resource managers play a vital role in computational systems by managing and allocating resources to tasks efficiently. They are essential for ensuring optimal system performance, particularly in environments where resources such as CPUs, GPUs, memory, and storage are shared among multiple users or applications. These systems aim to maximize resource utilization, minimize job completion time (JCT), and balance workloads to prevent bottlenecks or underutilization. Together, schedulers and resource managers enable computational systems to meet the demands of diverse workloads reliably and effectively [40].

Schedulers are primarily responsible for determining the execution order of tasks based on factors such as priority, resource requirements, and predefined system policies. This ensures minimal delays and prevents resource conflicts while maintaining fairness among users. Resource managers, on the other hand, handle the assignment of specific resources to tasks, considering their unique computational, memory, and latency needs. By coordinating their efforts, schedulers and resource managers address key challenges such as resource contention, dynamic workloads, and varying task priorities.

One of the significant challenges in managing resources is resource contention, where multiple tasks compete for the same limited resources. This requires intelligent allocation strategies to avoid performance degradation. Furthermore, tasks often have diverse and complex resource requirements, making it difficult to design a one-size-fits-all allocation approach. The dynamic nature of workloads in multi-tenant environments, such as cloud computing, adds another layer of complexity, as schedulers must adapt to fluctuating demands in real time. Dependencies among tasks also complicate scheduling, as some jobs rely on the completion of others, requiring careful coordination to prevent delays or deadlocks. Balancing fairness across users while adhering to priority policies poses another challenge, particularly in competitive resource environments. Finally, as the scale of tasks and resources grows, schedulers and resource managers must remain efficient and scalable to avoid becoming a bottleneck themselves.

The decision-making process in schedulers and resource managers involves several key steps, though it faces significant challenges. One of the major difficulties is

accurately estimating the runtime and resource requirements of tasks, both of which are critical for effective scheduling. Most schedulers rely on users to specify task properties, such as expected runtime and resource demands (e.g., memory and compute power). However, users often lack precise knowledge of these parameters or provide inaccurate estimates, which can lead to inefficiencies in resource allocation and scheduling decisions. Systems like SLURM (Simple Linux Utility for Resource Management) [41, 42] address this issue to some extent by requiring users to specify these details during job submission and allowing administrators to design partitions or queues tailored to specific job types. For example, partitions may be created to group tasks based on their expected runtimes (short, medium, or long) or their resource needs, enabling more predictable and efficient scheduling behavior. . However, SLURM places a significant burden on both users and administrators, as users must manually specify resource requirements upfront, and administrators must carefully design and maintain partitions to match workload characteristics. This static approach can lead to inefficiencies, particularly in dynamic workloads where resource demands fluctuate. To address this limitation, a more adaptive and intelligent resource management system is needed—one that can dynamically allocate resources based on real-time usage patterns rather than relying solely on predefined configurations. In Chapter 6, we envision our resource manager evolving toward integration with systems like SLURM, enhancing its capabilities with automated decision-making to reduce the manual effort required from users and administrators while improving overall efficiency.

Several widely-used systems exemplify the principles of schedulers and resource managers. Kubernetes [43], for example, is a container orchestration platform that automates the deployment, scaling, and management of containerized applications. It incorporates powerful scheduling features to allocate resources effectively across a cluster. Apache Mesos [44] provides a distributed systems kernel that abstracts resources like CPU, memory, and storage, enabling dynamic resource allocation for frameworks such as Hadoop [45] and Spark [46]. In high-performance computing environments, Slurm is a highly scalable resource manager that schedules and manages jobs across supercomputers. YARN (Yet Another Resource Negotiator) [45] serves as a resource management layer within the Hadoop ecosystem, dynamically allocating resources to applications based on their requirements. Similarly, Apache Hadoop MapReduce includes a scheduler that divides tasks into smaller subtasks and distributes them across systems for parallel processing.

2.4.1 High-Performance Computing (HPC) setup vs. Cloud

HPC and cloud computing are two distinct paradigms for delivering computational resources, each tailored to different types of tasks, requirements, and user expectations. HPC systems are designed for large-scale, compute-intensive tasks that require significant parallel processing power, such as scientific simulations, weather modeling, and genome sequencing. These tasks often run for extended durations, sometimes spanning days or weeks, and are highly sensitive to performance and resource availability. HPC environments prioritize low-latency, high-throughput performance, with

dedicated resources and tightly controlled Quality of Service (QoS) to minimize delays and optimize execution times [47]. On the other hand, cloud computing is more versatile, offering elastic and on-demand resources for a wide variety of applications, including web services, data analytics, and software development. Cloud tasks are often shorter in duration, designed for scalability and flexibility rather than sustained performance. Unlike HPC, which relies on dedicated infrastructure, cloud systems use shared resources, which can lead to variability in response times and QoS [48].

In terms of user experience, HPC systems are optimized for tasks where predictability and maximum performance are critical, but they typically require detailed planning, including predefined task durations and resource specifications. In contrast, cloud computing excels in providing ease of access, rapid provisioning, and pay-as-you-go pricing, making it more suitable for workloads that are dynamic and less performance-critical. While HPC is favored in scenarios where delay or downtime could compromise results (e.g., simulations or large-scale computations), the cloud is ideal for tasks where scalability and accessibility are more important than absolute performance consistency.

When considering deep learning workflows, training generally belongs to the HPC domain due to its high computational demands, long runtimes, and sensitivity to resource performance. Training deep learning models involves processing massive datasets through multiple epochs, requiring efficient GPU utilization, low-latency interconnects, and substantial memory resources—all of which are hallmarks of HPC environments. Conversely, inference aligns more closely with cloud computing, as it typically involves deploying trained models for real-time or near-real-time predictions. Inference tasks are lighter in terms of computational load, require lower latency for end-user interactions, and benefit from the scalability and global accessibility of cloud platforms. This division highlights how the unique characteristics of HPC and cloud computing make them complementary rather than interchangeable, serving distinct stages in the lifecycle of deep learning applications.

2.4.2 Scheduling and Mapping

Scheduling and mapping are two fundamental processes in resource management systems, each serving distinct but interconnected roles. Scheduling focuses on determining the order and timing of task execution, often based on policies such as priority, fairness, or first-come-first-served. It decides "when" a task will run, ensuring that tasks are executed in an orderly manner while adhering to system objectives, such as minimizing delays or maximizing throughput. Mapping, on the other hand, deals with assigning specific resources to tasks, deciding "where" a task will execute. This involves selecting the appropriate hardware resources—such as GPUs, CPUs, or memory—based on the task's requirements and the current system state. While scheduling ensures that tasks are queued and executed efficiently over time, mapping ensures that the allocated resources are suitable for the task's needs.

In this work, our primary focus is on the mapping side of resource management systems, with an emphasis on efficient collocation. Efficient collocation involves assigning multiple training tasks to the same GPU, in a manner that maximizes

resource utilization while minimizing performance degradation and out-of-memory (OOM) errors. This distinction between scheduling and mapping is critical, as effective mapping strategies enable resource managers to make better decisions about how to allocate computational power, particularly in multi-tenant and deep learning workloads. Our work seeks to push the boundaries of what can be achieved through thoughtful and efficient task mapping in modern computing environments.

2.5 Schedulers and Resource Managers for Deep Learning

The challenge of GPU scheduling/ resource management inefficiencies in deep learning (DL) workloads has led to numerous research contributions. Below, we categorize and summarize the key works addressing GPU resource management.

2.5.1 Empirical and Survey-based Studies on GPU Scheduling and Resource Management

Empirical analyses highlight inefficiencies in GPU scheduling at large scales.

- **MLaaS** [49] analyzes workload traces from Alibaba's production MLaaS cluster with over 6,000 GPUs, identifying inefficiencies such as low GPU utilization, long queuing delays, and scheduling challenges for high-end GPU tasks. The study proposes GPU sharing and a reserving-and-packing strategy to improve resource allocation but highlights open issues like load imbalance and CPU bottlenecks, calling for further research. While the study proposes strategies like GPU sharing and a reserving-and-packing scheduling policy to enhance resource allocation, it does not deeply analyze application-specific requirements to build predictive models for estimating resource needs. Additionally, the study does not monitor GPU memory and utilization to identify opportunities for task collocation.

- **Analysis of Multi-Tenant GPU Clusters** [12] analyze approximately 100,000 jobs over two months in Microsoft's GPU cluster. They find that gang scheduling and strict locality constraints contribute to resource contention and queuing delays. The study suggests that relaxing certain locality constraints and improving gang scheduling flexibility could mitigate these issues.

- **Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters** [50] analyzed large-scale SenseTime job traces [51], revealing key insights for cluster system design. Notably, 50% of GPU jobs complete within 10 minutes, multi-GPU jobs dominate utilization, and a small fraction of users consume most resources. The study introduced a predictive framework leveraging historical data for resource management. As case studies, they proposed Quasi-Shortest-Service-First scheduling, reducing average job completion time, and a Cluster Energy Saving service, improving utilization by up to 13%. However, their approach focuses only on a high-level system

perspective, does not model task-specific resource needs, and does not address collocation. In contrast, our work tightly integrates model-aware predictions with GPU monitoring and intelligent packing to enhance utilization and energy efficiency.

- **Deep Learning Workload Scheduling in GPU Datacenters: A Survey** [52] provides a comprehensive review of scheduling techniques for DL training and inference workloads. The survey categorizes existing schedulers based on objectives and resource consumption features, offering a taxonomy of current approaches.

- **Deep Learning Workload Scheduling in GPU Datacenters: Taxonomy, Challenges and Vision** [53] surveys existing research efforts for both training and inference workloads, presenting how existing schedulers facilitate the respective workloads from the scheduling objectives and resource consumption features.

2.5.2 Adaptive and Predictive Scheduling Techniques

Methods that use adaptive and predictive scheduling to improve GPU utilization.

- **Gandiva** [54] exploits the cyclic GPU memory usage of DL jobs for suspend-resume scheduling. It enables time-sharing and optimizes communication-intensive task placement. However, it relies on preemptive scheduling, which may not suit all workloads.

- **Salus** [55] introduces iteration-level scheduling, allowing fast suspend-resume mechanisms without kernel overhead. It enables fine-grained GPU sharing among multiple deep learning (DL) applications by implementing two key primitives: fast job switching and memory sharing. This approach improves GPU utilization and reduces job completion times.

- **Pollux** [56] dynamically reallocates resources using a goodput metric, optimizing per-job and cluster-wide efficiency. It introduces a co-adaptive scheduling approach that simultaneously adjusts resource allocation and training parameters, such as batch size and learning rate, to maximize goodput—a combination of system throughput and statistical efficiency.

- **Tiresias** [57] minimizes deep learning job completion times using Gittins index and Least-Attained Service (LAS)-based scheduling. It introduces a Two-Dimensional Attained Service-Based Scheduler (2DAS) that assigns priorities to jobs based on their attained service, calculated from the number of GPUs used and the elapsed running time. When job duration information is unavailable, Tiresias applies the LAS algorithm, inversely prioritizing jobs with less attained service. If job duration distribution is known, it utilizes the Gittins index to prioritize jobs likely to complete sooner. This approach effectively reduces average job completion times in GPU clusters.

- **E-LAS** [58] reduces deep learning training time by utilizing real-time epoch progress rate estimations. It improves upon existing schedulers by prioritizing jobs based on their current epoch progress, allowing for more informed scheduling decisions without prior knowledge of job characteristics. This approach enhances scheduling efficiency and reduces average job completion time.

The aforementioned work primarily focus on optimizing resource allocation through techniques like preemptive scheduling, fairness enforcement, fine-grained sharing, and dynamic resource reallocation. However, they often do not deeply analyze application-specific resource requirements or monitor GPU utilization metrics to build predictive models for estimating GPU memory needs. In contrast, our approach emphasizes developing such predictive models and closely monitoring GPU utilization to identify opportunities for task colocation, aiming to avoid out-of-memory errors and ensure system responsiveness across diverse workloads.

2.5.3 GPU Affinity-Aware and Isolation Strategies

Ensuring safe and efficient resource allocation through affinity-aware scheduling.

- **HiveD** [59] introduces hierarchical GPU affinity-aware scheduling to maintain fairness and prevent resource contention. It utilizes a multi-level cell structure to capture different levels of GPU affinity within a cluster, allowing for the creation of Virtual Private Clusters (VPCs) for each tenant. This design ensures that deep learning jobs are scheduled with the required GPU affinities, reducing queuing delays.

- **Vapor** [60] employs preemptive scheduling and adaptive batch redistribution to maximize GPU efficiency. It introduces two novel scheduling policies: preemptive GPU sharing and adaptive batch scheduling. Preemptive GPU sharing allows for the interruption and resumption of tasks to better utilize GPU resources, while adaptive batch scheduling dynamically adjusts the batch sizes of deep learning tasks to optimize both computation and communication efficiency.

- **Horus** [61] introduces interference-aware scheduling by proactively predicting GPU utilization of deep learning (DL) jobs based on their computation graph features. This approach allows for optimal job placement, minimizing performance degradation due to resource contention. Horus employs a coarse-grained GPU utilization metric, which does not capture fine-grained resource demands accurately. Additionally, the model it adopts for GPU memory estimation tends to over-predict usage, potentially leading to underutilization of resources.

- **Scheduling Deep Learning Jobs in Multi-Tenant GPU Clusters via Wise Resource Sharing** [62] proposes a heuristic-based GPU sharing model that allows multiple DL jobs to share the same set of GPUs without altering their training settings. The authors introduce the SJF-BSBF (Shortest Job First with Best Sharing Benefit First) scheduling algorithm, which intelligently selects job pairs for GPU resource sharing and determines runtime settings, such

as sub-batch size and scheduling time points. This approach aims to optimize overall performance while ensuring DL convergence accuracy through gradient accumulation.

However, these approaches often do not deeply analyze application-specific resource requirements or develop predictive models for GPU memory usage. Additionally, they may lack mechanisms for monitoring real-time GPU utilization to identify opportunities for task collocation, which are essential for avoiding out-of-memory errors and ensuring system responsiveness across diverse workloads. At the same time, their optimization scheduling techniques are orthogonal to our work and for further improvement, they can be part of the holistic system.

2.5.4 Fair and Cost-Efficiency in Scheduling

Approaches ensuring fairness and cost-efficient GPU resource allocation.

- **Themis** [63] enforces fairness using a two-level bidding system, dynamically allocating GPUs while balancing efficiency and fairness. It introduces the concept of finish-time fairness, ensuring that machine learning workloads complete in a manner proportional to their fair share of resources.

- **AlloX** [64] models scheduling as a min-cost bipartite matching problem, ensuring dynamic fair allocation. It addresses the challenge of allocating interchangeable resources, such as CPUs and GPUs, in hybrid clusters by transforming the scheduling problem into a min-cost bipartite matching framework. This approach allows AlloX to provide dynamic fair allocation over time, optimizing performance while maintaining fairness among users.

- **Cynthia** [65] predicts training time using a lightweight analytical performance model, optimizing cloud-based GPU provisioning. It addresses the unpredictable performance of distributed deep neural network (DDNN) training in cloud environments by considering factors such as resource bottlenecks, heterogeneity, and the imbalance between computation and communication. By leveraging resource consumption data from workers and parameter servers, Cynthia accurately predicts training performance and provisions cost-efficient cloud instances to meet specific training time and loss objectives.

As the techniques proposed in this section have a focus on fairness and cost-efficiency through scheduling techniques, they are orthogonal to this thesis and complement each other if combined.

2.5.5 Hyperparameter Tuning and Job Scheduling Coordination

Methods integrating hyperparameter tuning with scheduling mechanisms.

- **Fluid** [66] optimizes hyperparameter tuning by coordinating evaluation trials with cluster resources through a water-filling scheduling approach. This method

enhances resource utilization at both intra- and inter-GPU levels, accelerating tuning processes.

- **Gavel** [67] generalizes scheduling policies to account for hardware heterogeneity. It systematically transforms existing scheduling policies into heterogeneity-aware versions using an abstraction called effective throughput. This approach allows Gavel to optimize various objectives, such as fairness and makespan, in heterogeneous clusters.

2.5.6 Optimization-Based Scheduling Approaches

These methods transform scheduling into optimization problems for efficiency.

- **AITurbo** [68] distinguishes between predictable and unpredictable jobs while unifying CPU-GPU allocation. It introduces a novel resource scheduler that treats predictable and unpredictable jobs separately, allocating heterogeneous CPU-GPU resources in a unified manner. Predictable jobs exhibit consistent performance metrics across different runs, while unpredictable jobs show significant variability due to factors like dynamic data loading or varying computational patterns. To predict the performance of predictable jobs, AITurbo employs performance models that estimate training times and resource utilization under various CPU-GPU allocations. By predicting model performance under various resource allocations, AITurbo optimizes scheduling decisions to enhance overall system efficiency.

- **Optimus** [69] minimizes training time by dynamically adjusting resources based on online models. It introduces a customized job scheduler for deep learning clusters, which utilizes online resource-performance models to predict training speed as a function of allocated resources. By adjusting the number and placement of workers and parameter servers during runtime, Optimus maximizes resource efficiency and training speed.

- **Prophet** [70] optimizes gradient transfer scheduling to enhance GPU and network utilization. It introduces a predictable communication strategy that organizes gradient transfers in an optimal sequence, aiming to maximize resource utilization during distributed deep neural network (DDNN) training. By leveraging the observed stepwise pattern of gradient transfer start times, Prophet predicts the appropriate number of gradients to group into blocks and schedules their transfer to maintain high GPU and network utilization. This approach reduces GPU idle time and accelerates the training process.

2.5.7 Reinforcement Learning-Based Scheduling

Schedulers employing reinforcement learning to optimize GPU job scheduling.

- **DL2** [71] applies reinforcement learning to dynamically adjust GPU resource allocation over time. It introduces a deep learning-driven scheduler for deep

learning clusters, aiming to expedite global training jobs by dynamically resizing resources allocated to jobs. DL2 employs a combination of supervised learning and reinforcement learning to optimize resource allocation decisions, thereby improving overall cluster efficiency.

- **Harmony** [72] employs deep reinforcement learning to optimize job placement in distributed machine learning clusters, aiming to minimize interference and enhance performance. By implicitly encoding workload interference within a neural network, Harmony maps cluster and job states to placement decisions, reducing performance unpredictability caused by resource contention among co-located jobs.

2.5.8 Industry-Deployed Scheduling Solutions

Solutions used in production environments for large-scale DL training.

- **AntMan** [73] is Alibaba's deep learning framework that co-executes jobs on GPUs and dynamically scales resources. It introduces dynamic scaling mechanisms within deep learning frameworks, allowing for fine-grained coordination between jobs and preventing interference. However, integrating AntMan necessitates modifications to existing deep learning frameworks and cluster schedulers, potentially increasing system complexity and maintenance efforts. Additionally, co-executing multiple jobs on shared GPUs can lead to performance interference between tasks, affecting individual training efficiency. While AntMan has been successfully deployed in large-scale production environments, scaling the system to accommodate an even larger number of jobs and GPUs may present challenges in maintaining performance and efficiency.

- **FfDL** [74] is IBM's deep learning platform that balances dependability with scalability, elasticity, flexibility, and efficiency in cloud-based DL training. It supports multiple DL frameworks, such as TensorFlow, Caffe, and PyTorch, and offers features like multi-tenant resource sharing, dynamic job scheduling, and fault tolerance. FfDL enables users to train models at scale while efficiently managing underlying resources.

2.5.9 Advanced Techniques for DL Model Execution Optimization

Methods for optimizing DL model execution for scheduling improvements.

- **SAD** [75] optimizes CPU allocation for GPU-based deep learning jobs using adaptive inference. It introduces a performance predictor that accurately suggests training speeds for different CPU counts across various GPUs, enabling efficient CPU resource allocation. Preliminary results indicate that SAD can effectively balance CPU and GPU workloads, enhancing overall system performance. It is orthogonal to the work of this thesis and can be complementing as it brings the CPU awareness into the consideration.

- **OOO Backprop** [76] improves GPU utilization by reordering gradient computations to minimize stalls. By exploiting the dependencies of gradient computations, OOO Backprop enables reordering their executions to make the most of the GPU resources. This approach enhances GPU utilization in single-GPU, data-parallel, and pipeline-parallel training by prioritizing critical operations and reducing idle times.

To conclude, work related to scheduling is orthogonal to this thesis, as the primary focus here is on efficient collocation, supported by accurate GPU memory estimation for deep learning tasks, while monitoring GPUs using a representative GPU utilization metric.

2.6 Open Challenges

GPUs lack support for a fine-grained hardware resource-sharing method among different tasks [55, 77, 78, 79]. Software GPU sharing techniques cannot be candidate solutions due to their large overheads [80, 81]. Thus, the GPU under-utilization problem accompanies DL training tasks [12, 82]. When it comes to streams, MPS, and MIG, identifying the correct combination of them for a given deep learning task is still an open challenge.

Collocating, also known as overprovisioning, unrelated tasks, especially on distributed training tasks, can further lower the utilization of GPUs. This can be due to **network overhead** (in cases when there are several clusters) and **interference**.

MIG puts forward a coarse-grained sharing of GPUs. However, some tasks cannot saturate their assigned GPU instance. Mixing tasks can be an efficiency-rewarding idea [83]. Using the MPS mechanism over MIG, with interference consideration of collocating jobs, can result in substantial utilization improvements.

Distributed training (due to large data sets) often requires the use of multiple GPUs [84] and ML frameworks require training tasks on each GPU to be scheduled at the same time, i.e., gang scheduling [85]. This increases the risk of resource fragmentation, and low utilization in shared clusters of GPUs [12]. In distributed training, frameworks like TensorFlow train models with Synchronous Stochastic Gradient Descent (S-SGD). They process a batch of data partitioned across GPUs simultaneously and average the resulting gradients to obtain an updated global model. The common way to increase the utilization of GPUs is to increase the batch size, although it may result in lower statistical efficiency. The common solution is tuning hyper-parameters like learning rate, which is complex and model-specific [86].

Furthermore, DL training jobs are based on a trial-and-error mechanism, such as hyper-parameter search, which can be manual or automated. Users usually try several configurations and use early feedback to decide whether to prioritize or kill a subset of them [54, 87].

Using traditional schedulers, such as Apache Yarn [45] and Kubernetes [43], which are fit for big-data processing, leads to **head-of-line-blocking**. This is due to

the fixed and exclusive scheduling policies (non-preemptive scheduling of arriving jobs) adopted by the aforementioned schedulers. The exclusive assignment of GPUs to jobs on their startup and waiting until their completion while queuing other jobs (ending in a long queuing experience) is the obvious downside of these schedulers for the DL training. Additionally, they consider jobs as black boxes [88, 89] and do not consider jobs' behavior or characteristics. This negligence can end in resource interference and more under-utilization [54, 57, 87, 63, 90]. On the other hand, the overhead of profiling-based methods are not affordable and reliable as the behavior of a task can change over its execution.

Chapter 3

Profiling and Monitoring Deep Learning Training Tasks

Deep learning training tasks (DLTTs) require significant hardware resources, making their optimization crucial for sustainable development and continuity [91, 12]. Achieving this optimization necessitates a deep understanding of workload behavior and resource requirements, particularly regarding computing power and memory usage. Strategies may include refining models and training processes, reducing operations, minimizing memory usage, or implementing improvements at the resource-manager or scheduler level, such as orchestration and resource overprovisioning. To facilitate these optimizations, it is essential to profile the workloads and efficiently monitor the infrastructure they utilize. We identified, studied, and experimented with various profiling and monitoring tools, benchmarking their benefits, drawbacks, and costs to determine the appropriate contexts for their use. Our key insight is that monitoring tools are well-suited for real-time decision-making, while profiling tools offer developers a deeper understanding of code execution at the hardware level, albeit with higher costs in terms of time, computation, and storage. Additionally, we evaluated commonly used metrics to assess their effectiveness and emphasized the importance of selecting representative metrics for monitoring computing devices. Monitoring tools should be leveraged for online decision-making, while profiling tools are best reserved for deeper or system-wide optimizations. It's also important to carefully manage the volume of data collected through these processes. The primary objective of our study was to identify tools that provide timely and actionable information. Our findings reveal that tools like top, nvidia-smi, and dcgmi impose minimal overheads in terms of time, GPU, and CPU usage, making them ideal for real-time decision-making systems. In contrast, the substantial overheads associated with Nsight Systems and Nsight Compute render them impractical for such scenarios. Moreover, the metrics provided by the latter one focus on kernel-level performance, which may not align with application-level needs.

3.1 Introduction

Deep learning training requires high computing and memory resources and is a highly parallel process. This has naturally lead to accelerating the training processes with hardware architectures such as GPUs that can exploit these traits. On

the other hand, matching the computing and memory requirements of deep learning training to the capabilities of modern GPUs is not straightforward for all deep learning applications and GPU types. This mismatch results in slowdowns and resource underutilization [92, 82, 93, 86]. To find solutions to these challenges, it is essential to characterize the interaction between deep learning systems and their underlying hardware. This is obtainable by profiling systems and monitoring hardware utilization.

Profiling provides the developers with insights in how the application behaves in terms of computing and memory patterns and requirements. Afterwards, the data and trace plots can assist in finding and addressing the bottlenecks. Monitoring tools, in contrast, reveal how specific hardware resources react to the execution of applications. One can find out whether the current configuration of the model training saturates the hardware resources. The workload can be scaled up or more applications may be run simultaneously to optimize for both high utilization and training performance.

Using profiling and monitoring tools effectively is an art and can be time-consuming for beginners. Furthermore, in the field of deep learning, one has to understand the tools for not only CPUs but also accelerators like GPUs. While there are many works utilizing tools for CPUs (e.g., top, perf, Intel VTune) for workload characterization [94, 95, 96, 97, 98, 99, 100, 101], tools for accelerators are less mature and rapidly evolving, and relatively unexplored. To address this challenge, this paper reviews the most relevant profiling and monitoring tools for deep learning workloads. We investigate the strengths and limitations of the profiling tools offered by NVIDIA, Nsight Systems and Compute, in addition to the monitoring tools `nvidia-smi` and `dcgm`. We do this by (1) surveying the functionality offered by these tools, (2) studying the metrics reported and showing the shortcomings of widely used high-level utilization metrics, and (3) measuring profiling and monitoring tools' overheads while running both light and heavy deep learning training scenarios.

Our investigation demonstrates the following:

- The negligible overhead of the monitoring tools make them ideal candidates to be integrated into task schedulers and resource managers for online decision-making.
- On the other hand, the GPU utilization and GRACT metrics offered by `nvidia-smi` and `dcgm`, respectively, are too high-level and unrepresentative for actual GPU utilization. More concrete metrics such as SMACT and SMOCC from `dcgm` may help to overcome this issue.
- The profiling tools are effective for targeted code optimizations, but their overheads make them unsuitable for online decision making. The profiling mode of Nsight Compute in particular heavily disrupts a training run.
- Each profiling tool has their time to shine. Profiling tools integrated into deep learning frameworks and Nsight Systems offer a way to detect bottlenecks with application-specific and system-wide views, respectively. However, to further optimize individual kernels, a tool like Nsight Compute offer deeper insights at the micro-architectural level of a GPU. Thus, one can create a pipeline of profiling

stages using a mix of tools.

3.2 Tools

This section surveys the most relevant profiling and monitoring tools for deep learning training on NVIDIA GPUs.

3.2.1 Profiling Tools

There is a range of tools available to profile deep learning workloads. Tools integrated with the deep learning frameworks, such as the TensorFlow and PyTorch profilers [102, 103], are immediately available to those using their respective frameworks. Alternatively, NVIDIA provides the profiling tools Nsight Systems and Nsight Compute. This section goes over the PyTorch profiler, as a representative framework tool, and the NVIDIA profiling tools.

The PyTorch Profiler [103] is a trace-based profiling tool that can automatically collect a range of performance metrics during both deep learning training and inference. As it is integrated into the deep learning framework itself, running the profiler is just a matter of adding a few lines of Python code. It requires less setup than other monitoring or profiling tools due to being specific to PyTorch and deep learning. Being integrated into the code, the profiler allows for extensive control of which iterations are profiled. This prevents the profiling data from growing out of hand. It is in fact recommended that the users profile one or more iterations in an epoch rather than whole epochs, since the behavior of the iterations over each batch tends to be repetitive. In 3.3, we highlight this while discussing the overheads of the PyTorch profiler.

NVIDIA Nsight Systems [104], `nsys`, is a trace-based profiler similar to the PyTorch profiler. It constructs a timeline of CPU and GPU events. This notably includes different compute and memory access streams on the GPU, yielding valuable information such as data movement bottlenecks and most frequently used kernels. `nsys` is framework-independent and can effectively profile a variety of software. Furthermore, it offers a system-wide view, including more insights to interactions with the operating system and network compared to the more application-focused view given by the framework profilers. `nsys`, thus, does not annotate the deep learning traces out of the box. NVTX, NVIDIA Tools Extension [105], provides an API to enable annotating the training code itself. Multiple deep learning libraries, including PyTorch [28], support NVTX annotations in their code.

`nsys` runs as a separate process while profiling an application. Applications can be profiled both online/interactive and offline. The profiling is done either via a GUI or a command line with the level of detail specified by the user. For example, a user can launch `nsys` to track the GPU memory usage by kernels, enable the collection of backtraces, and collect metrics from network interface cards.

Bottlenecks can be detected by viewing the timeline of computing and memory operations. For example, a timeline detailing that 90% of the time is spent on compute indicates that the workload is compute-intensive and that compute-side optimizations might improve the application. Conversely, when there are a lot of data access stalls, the workload is memory-intensive and improved data orchestration may greatly improve runtime.

It should be noted that `nsys` does not work when multi-instance GPU (MIG) mode [36, 39], which divides a GPU into smaller instances, is enabled on any GPU on the server. While this is a current functional limitation as MIG technology is relatively new and has been maturing, it may be fixed over time. In addition, carelessly specifying more and more profiling options to get more details can result in longer post-processing times after the profiling is over and bigger trace files that are harder to render in the tool's GUI.

NVIDIA Nsight Compute [106], `ncu`, allows for in-depth GPU analysis. It disrupts the regular run of a program and reruns the kernels of a program multiple times to trace the micro-architectural behavior.

Similar to `nsys`, `ncu` has a GUI-based and command line interface, where the users can specify the amount information to trace. As the nature of the profiling is disruptive, it is often run as an online interactive profiler and debugger, though offline mode is also supported. Compared to the PyTorch Profiler and `nsys`, the main strength of `ncu` is the degree of detail and granularity it provides when profiling. It illustrates the data movement behavior across the different levels of the GPU memory hierarchy and helps to identify data stalls in kernels. It also maps the metrics to the individual lines of code that contribute to them by connecting assembly (SASS) code with parallel thread execution instruction set architecture (PTX or NVPTX), an architecture independent intermediate representation for CUDA, and with high-level code (e.g., CUDA, C/C++, Fortran, OpenACC, Python). Additionally, it can export CUDA execution graphs and allow the profiling of individual nodes in these graphs.

While `ncu` is good to investigate things at a microscopic level, it does impact application behavior. Its profiling depends on the principle of rerunning kernels multiple times either one kernel at a time (*kernel mode*) or via iterating over the application multiple times (*application mode*) as specified by the user. In each iteration, additional data for the target kernel(s) is collected. Application replay requires the program execution to be deterministic.

While `ncu` provides extremely detailed information at the kernel level, it is often difficult to map the information to the application level. Furthermore, as a result of the repetitive kernel runs, the profiling overhead on the application is very high. Therefore, `ncu` should mainly be used to optimize individual kernels, not application-level scheduling behavior. Since it supplies the users with GPU architecture and micro-architecture-related information, it is extremely useful for computer architects and low-level library developers.

3.2.2 Monitoring Tools

There are a variety of monitoring tools for servers to observe their utilization behavior such as how many CPU cores are in use, how many GPU streaming multiprocessors are active, what the CPU/GPU memory consumption is, etc. Such observations can aide cluster administration, hardware resource management, and workload scheduling decisions, even in real-time thanks to the low-overhead of the monitoring tools (as quantified in 3.3.2). On the other hand, in contrast to the profiling tools, these tools cannot be used for coming up with optimization ideas for a specific application's internals or kernels. In this section, we cover the two monitoring tools offered by NVIDIA: `nvidia-smi` and `dcgm`.

NVIDIA System Management Interface [107], `nvidia-smi`, provides monitoring and management capabilities for NVIDIA GPUs. Users can interact with it via command line to (1) configure a GPU's performance parameters like changing the frequency, setting power cap, etc., (2) set the preferred multi-instance GPU (MIG) partitions, and (3) track a range of performance metrics such as GPU utilization, size of GPU memory usage, performance state and temperature of a GPU, etc. One can view the metrics tracked via standard output or write them to a CSV or XML file. These metrics can be tracked system-wide, for a GPU, and for an application.

Underneath, `nvidia-smi` uses the NVIDIA Management Library (NVML) [108], which provides an API for monitoring and managing various states of NVIDIA GPUs. NVML provides direct access to the queries and commands that enables the monitoring done by `nvidia-smi`. If users want to customize the monitoring, they can write a custom program using NVML instead of using what is exposed by `nvidia-smi`.

While `nvidia-smi` helps with basic system monitoring, it is still limited in terms of the metrics it provides. For example, it doesn't track the interactions between the CPU and the GPU. Furthermore, on a MIG-enabled GPU, it tracks metrics from the whole GPU and not from individual MIG instances.

NVIDIA Data Center GPU Manager [109], `dcgm`, provides more detailed information about hardware utilization on CPU-GPU co-processors compared to `nvidia-smi`. `dcgm` can ease the management and configuration of GPUs in a cluster by providing features such as GPU grouping. Furthermore, it can track not just high-level GPU utilization, but also occupation and activity of streaming multiprocessors and utilization of tensor cores. It can give more detailed insights on energy consumption of the GPU and the data movement across CPU and GPU and different GPUs by reporting how much the PCIe / NVLink bandwidth is used. Finally, it can also monitor the utilization of the individual MIG instances. On the other hand, since both `ncu` and `dcgm` use the same hardware counters underneath, they cannot be used simultaneously.

TABLE 3.1: Specifications of an A100 GPU - 40GB

Property	**Value**
GPU Architecture	NVIDIA Ampere
Compute Capability	8.0
#SMs	108
FP32 per SM	64
Tensor Cores per SM	4
Share Memory and L1 cache	192KB combined, Shared Memory is configurable up to 164KB
Max 32-bit Registers per SM	64KB
L2 cache	40MB
Memory	40 GB of high-speed HBM2 memory
Max Threads per Warp	32
Max Thread Blocks per SM	32
Max Warps per SM	64
Max Thread Block Size	1024
Max Registers per Thread	255

3.3 Experiments

After the qualitative overview of the tools of interest in 3.2, this section quantitatively analyzes them. We aim at answering the following questions with our experiments:

- What is the granularity of information reported by the different GPU utilization metrics?
- How intrusive are these tools on the execution of a deep learning training process?
- How much hardware resources do these tools need?
- How does the relative impact of these tools change based on the size and complexity of the deep learning training?

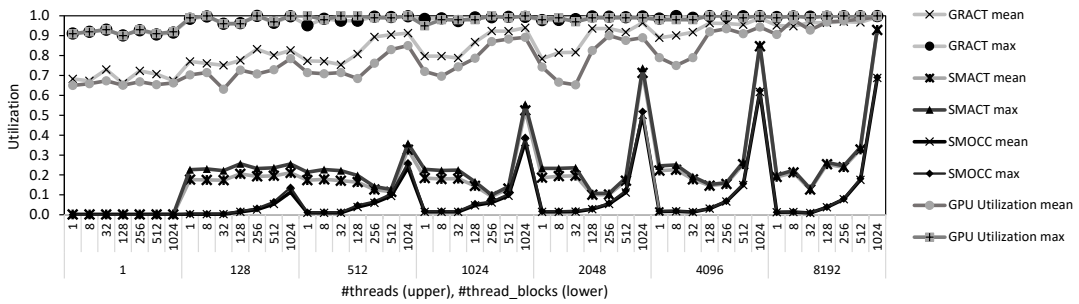


FIGURE 3.1: Different GPU utilization metrics as the load on the GPU varies.

3.3.1 Setup

All the experiments are run on a DGX Station A100, which is composed of an AMD EPYC 7742 CPU with 512GB of main memory and four A100 GPUs with 40GB of memory each. 6.1 details the specifications of the A100 GPU.

The system runs *DGX OS*, a variant of *Ubuntu 20.04.4 LTS*, and the installed CUDA version is 11.6.1.

Except for 3.3.2, the experiments are based on two types of training runs: (1) a *light* use-case with a simple CNN-model [110] trained on the MNIST [111] dataset, (2) a more *heavy* use-case in both computation and memory consumption with ResNet50 [112, 113] trained on the ImageNet dataset [1]. All models use the PyTorch framework, version 1.13.1, and are trained for 5 epochs. For the experiments in 3.3.2, where the different GPU utilization metrics are investigated, we create our custom micro-benchmark, which is described in the corresponding section.

We scoped down the experiments to a single light and heavy training scenario rather than experimenting with a wider variety of model training. When it comes to identifying the overhead caused by the tools, there can be two types of overhead: (1) fixed one such as fixed startup or shutdown overhead and fixed background resource usage, and (2) the one that vary based on the program complexity such as increased resource consumption due to more data being collected. The former would be more pronounced for the *light* training scenario, whereas it would be amortized or insignificant for the *heavy* scenario. The latter would be more significant for the *heavy* scenario. We argue that the overall conclusions for the respective behavior for these two types of overhead do not change across different light and heavy scenarios. This aligns with our experience using different training use cases with these profiling and monitoring tools.

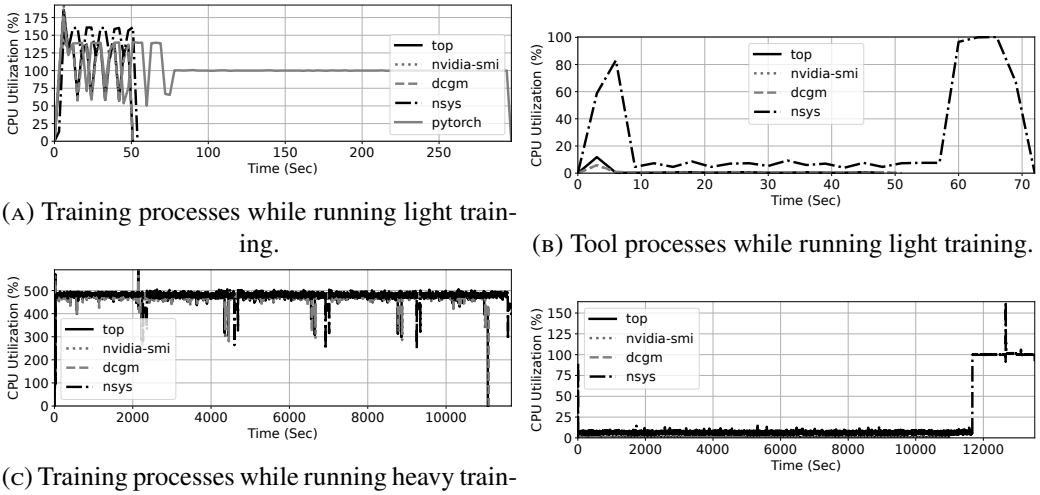
For the profilers, we run our experiments offline with the default settings. The PyTorch Profiler, `pytorch`, records both CUDA and CPU activity by default, while all of the extra options, such as flop estimation, are disabled. Running this with the 5-epochs of ResNet50 leads to prohibitive tracing information. Therefore, we only report `pytorch` results for the light training scenario. The traces collected by `nsys`, version 2022.1.3, are for CPU, CUDA, NVTX, OSRT, and OpenGL calls, as well as high-level resource utilization, but without CUDA backtracing. Finally, as `ncu`, version 2022.2.1, is by design a disruptive profiler (3.2.1), we decided that it is not insightful to report its overheads.

The raw experimental data can be found in our repository.[1]

3.3.2 Results

Among the questions listed above, 3.3.2 answers the first one, and 3.3.2 answers the remaining three.

[1] https://github.com/Resource-Aware-Data-systems-RAD/PMDLT



(A) Training processes while running light training.

(B) Tool processes while running light training.

(C) Training processes while running heavy training.

(D) Tool processes while running heavy training.

FIGURE 3.2: CPU Utilization.

GPU utilization

The popular metrics of interest while monitoring GPUs tend to be compute utilization, memory consumption, data movement, and energy consumption. Especially the GPU utilization can potentially be confusing due to the different ways for measuring this activity.

The GPU monitoring tools `nvidia-smi` and `dcgm` both have a *GPU utilization* metric, which is roughly defined as the % of time one or more kernels were executing on the GPU over the past sampling period. In addition, `dcgm` has multiple metrics to track the utilization of streaming multiprocessors (SMs). Most notable ones among these metrics are `GRACT`, `SMACT`, and `SMOCC`, which we investigate in this section.

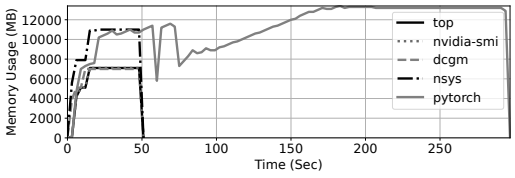
`GRACT`, *graphics engine activity*, is the fraction of time during which any portion of the graphics (e.g., ray tracing units) or compute engines were active. While `GRACT` tends to closely follow *GPU utilization* in values, in practice it is measured differently (sampling, hardware counters, etc.). Therefore, its over time values aren't exactly the same as what *GPU utilization* reports. `SMACT`, *SM activity*, refers to the fraction of active time on an SM, averaged over all SMs. Finally, `SMOCC`, *SM occupancy*, is the degree of parallelism within an SM (calculated by the unit of a warp, which typically occupies 32 threads in a thread block) relative to the maximum degree of parallelism supported by the SM.

For our investigation, we devise a micro-benchmark in which we vary the number of thread blocks and threads within a thread block in a kernel.[2] Each thread fetches a data item and calculates its square. 3.1 shows the results. As the figure highlights, even when there is only one thread within a single thread block, `GRACT` can show a utilization of 90%. This is extremely misleading when one is interested in whether the SMs are in use. In contrast, `SMACT` and `SMOCC` reveal substantially more information on SM utilization.

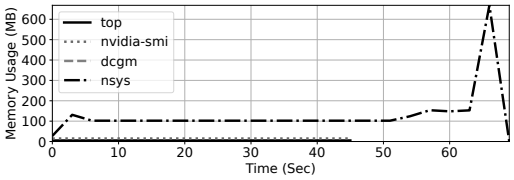
[2] https://github.com/Resource-Aware-Data-systems-RAD/PMDLT/blob/main/benchmark/square.cu

TABLE 3.2: Average epoch time w/o profiling and monitoring, and size of the information collected by the tools.

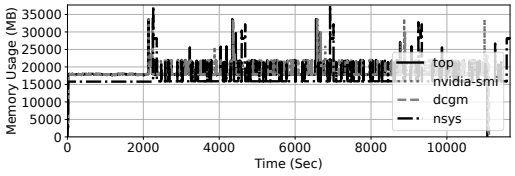
Tools	**simple CNN**		**ResNet50**	
	Time	Space	Time	Space
no tool	9.61sec	NA	37.06min	NA
top	9.66sec	∼20KB	37.11min	∼2MB
nvidia-smi	9.61sec	∼20KB	37.04min	∼2MB
dcgm	9.68sec	∼85KB	37.19min	∼8MB
nsys	9.88sec	∼40MB	39.13min	∼5GB
pytorch	13.65sec	∼1.4GB	NA	



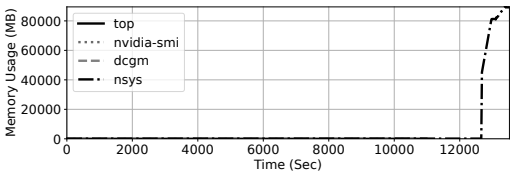
(A) Training processes while running light training.



(B) Tool processes while running light training.



(C) Training processes while running heavy training.



(D) Tool processes while running heavy training.

FIGURE 3.3: CPU memory usage.

The GPU used in our experiments (6.1) have support for up to 3456 thread blocks in total, where each thread block has support for up to 1024 threads in total. We see that `SMACT` and `SMOCC` reach higher values as we get closer to the limits of parallelism offered by the GPU, reflecting the actual load on the SMs. Due to possible overheads in orchestrating the threads, though, neither `SMACT` and `SMOCC` reach 100%.

In conclusion, one must be careful about the GPU utilization metrics to monitor depending on the goal of monitoring. For example, for a task scheduler that aims to decide which tasks to collocate on a GPU, plainly looking at GPU utilization or `GRACT` will miss the opportunities for utilizing the GPU better.

Tool Overheads

To quantify the overheads of the tools described in 3.2, we measure the epoch execution time, the size of the information produced by each tool, and utilization of CPU and GPU resources with and without using a particular tool. Since `top` [114] is used to collect CPU utilization information, we also include it in the results.

Execution time. To reason about the runtime impact of each tool, 3.2 reports the average epoch time. The execution time for each epoch is taken from PyTorch. As

3.2 shows, while the monitoring tools have negligible impact on execution time, the profiling tools lead to a noticeable overhead. 3.2 reveals that there is bigger runtime overhead for the profiling tools after the training is over, which is for post-processing the gathered information, omitted from 3.2. The overhead of the `pytorch` profiler is larger, since it collects more data by default compared to `nsys`.

Overall, while the monitoring tools can be integrated into online decision making, the profiling tools should be used for deliberate targeted investigations and optimizations.

Size of data files. 3.2 also reports the size of the information collected by each tool. For the monitoring tools, we manage how this information is stored, we simply write the information to an output file. As expected, the information collected by these tools have negligible overhead, and since `dcgm` offers more metrics to collect, it accumulates more data. The size of the information is larger for the profiling tools, since they collect more information. *pytorch* logs the actions that are part of the framework in great detail by default. In particular, the detailed stack traces of PyTorch functions and libraries contribute greatly to the scale of information. Additionally, the files are saved in the Chrome JSON format, which is not optimized for space at all. In comparison, *nsys* defaults to logging more generic system parameters as it is an application-independent solution. In addition, the files are saved in a compressed binary format. However, increasing the information collection in `nsys` would naturally increase the complexity and size of the trace files as well.

We highlight that while building a platform for systematically benchmarking the interaction between the deep learning applications and hardware, keeping all the monitoring and profiling information from various experiments may become a scalability challenge that has to be addressed.

CPU utilization. 3.2 shows the CPU utilization for both the training process itself while running a variety of profiling and monitoring tools and the processes created by the tools. We use `top` to report CPU utilization for each tool. Therefore, the line marked as `top` represents the baseline, and the rest of the tools run in parallel with `top`.

In 3.2a, we see that the monitoring tools have no visible impact on the CPU usage of the training process, since they don't increase the CPU utilization beyond the baseline. That is why the lines for `top`, `nvidia-smi`, `dcgm`, and `pytorch` overlap completely till around 50 seconds. The 50 seconds mark the training time for 5 epochs (3.2), where both the training and monitoring stops. On the other hand, `nsys` increases the CPU utilization slightly, $\sim$18%, while `pytorch` keeps it similar to the baseline training. However, `pytorch` has its post-processing phase performed by the main training process as well, which is why 3.2a depicts a lengthy 1-core utilization after the 5-epoch training is over for `pytorch`. As we can see in 3.2b, `nsys` launches a separate process for this purpose, which goes through the phases of (1) initialization (initial jump to $\sim$ 80%), (2) waiting for the training to be over (low utilization), (3) post-processing (brief $\sim$ 100% utilization). The post-processing time is shorter for `nsys` compared to `pytorch`. This is likely due to the

larger trace gathering performed by `pytorch` with its default settings compared to `nsys`. (see 3.2). Finally, the monitoring tools also spawn their own helper processes with negligible CPU utilization as we see in 3.2b.

3.2c and 3.2d show that in a heavier training scenario the impact of all tools on CPU utilization is insignificant, while the impact on the total execution time and post-processing is still visible with `nsys` finishing later.

Overall, when it comes to the profiling tools, one has to be mindful about the hardware resource consumption and the execution time of the post-processing phase of the profilers.

CPU memory usage. 3.3 shows the CPU memory usage for both the training process itself while running a variety of profiling and monitoring tools and the processes created by the tools. We use `top` once again to report the CPU memory consumption for each tool. Thus, the `top`-line represents the baseline similar to 3.2.

In 3.3a, we see that while the monitoring tools have no visible impact on the CPU memory usage of the training process, the profiling tools have an impact. Both `nsys` and `pytorch` increase the CPU memory usage (∼55%), and `pytorch`'s post-processing increases the total memory consumption further. In 3.3b, the `nsys` helper process has the same three-stage behavior as found in 3.2b. Finally, for the heavy training scenario, 3.3c reveals that the impact on resource usage is negligible for all of the tools during the training epochs. However, the impact of post-processing of the profiling tools is still considerable.

Overall, 3.3 exhibits similar trends to 3.2.

GPU resources. 3.4 shows the impact of the tools on the GPU resource usage. None of these tools create a separate helper process on the GPU. Therefore, the results are only for the training process. We retrieve these metrics from `dcgm`, which makes the bars for `dcgm` our baseline. As the GPU compute utilization metrics we report `SMACT` and `SMOCC` based on the results of 3.3.2. For the GPU memory utilization we use `DRAMA`, showing the frequency of memory accesses.

In general, when it comes to the impact on the GPU resource usage, all of the tools have negligible impact.

3.4 Related Work

While we study the main monitoring tools of NVIDIA GPUs, AMD offers similar tools such as ROCm-smi [115]. In addition, there are tools built on top of existing NVIDIA libraries. For example, `nvtop` [116] is a wrapper around NVML that provides visualization for NVML metrics, and Moneo [117] is a monitoring system that specifically targets AI applications.

Orthogonal to Nsight Systems and Nsight Compute, there has also been efforts to build profiling tools using the NVIDIA CUDA Profiling Tools Interface (CUPTI

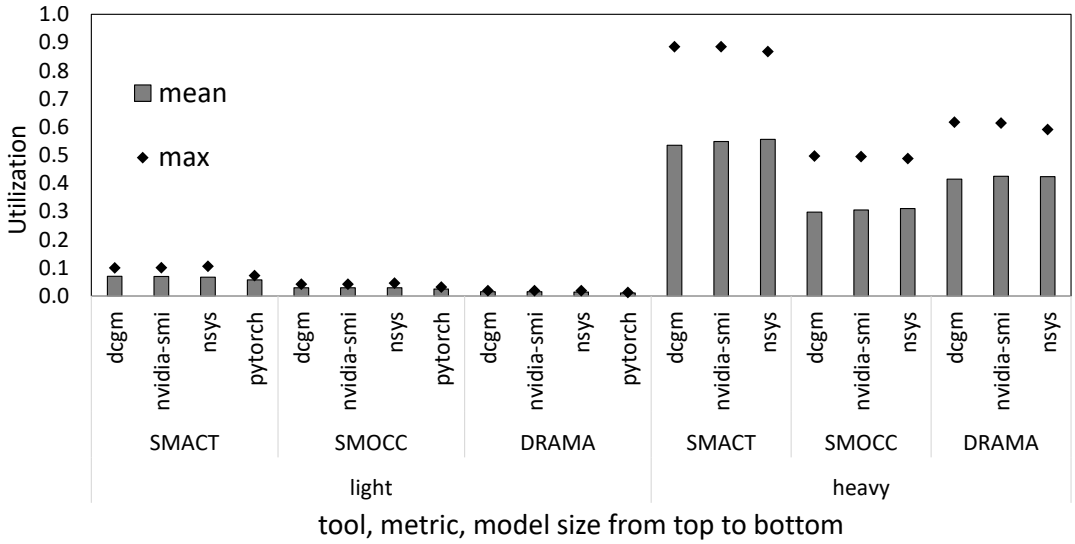


FIGURE 3.4: GPU utilization.

API [118]) [119, 120, 121, 122, 117, 123]. Furthermore, there are tools that have a stronger focus on profiling the data movement, such as [123], which is also built on top of CUPTI in addition to OSU INAM [124]. Finally, `nvprof` was also a tool for profiling on NVIDIA GPUs, but it has been deprecated by the release of Nsight Systems and Nsight Compute.

In this paper, we scoped our study to the most relevant tools provided by NVIDIA, while using the PyTorch profiler as a point of comparison, but a similar investigation can be done for the aforementioned tools using our methodology.

3.5 Conclusion

Deep learning models have become essential in many application domains but are expensive to train. It is thus important to understand the behavior of the training software and the underlying hardware. In this paper, we have analyzed the impact of monitoring and profiling tools on deep learning training. We have found that monitoring tools have negligible overhead and can be used for online decision making. In contrast, profiling tools offer more detailed information but incur time, space, and hardware resource consumption overheads. Additionally, one should be careful with their choice of metrics to monitor, as some paint a clearer picture than others, especially in the case of GPU utilization.

Profiling and monitoring tools have a fast and an ever-evolving nature. For instance, in the recent past, `dcgm` didn't report metrics for the `4g.20GB` MIG instance, but now it does. Similarly, the `NVML` library, which underlies `nvidia-smi`, recently added finer-grained `dcgm` metrics like SMACT and SMOCC. However, these additions are supported only on the newer NVIDIA GPUs such as the ones based on the Hopper architecture; one generation later than the Ampere architecture used in this study. This addition means that one can simply collect metrics using `nvidia-smi` if the point of interest is overall GPU utilization on the latest and emerging NVIDIA GPUs,

removing the dependency on multiple tools. Therefore, one should pay attention to using the up-to-date version of the tools on a given processor to determine the most effective subset of tools for a particular study.

Chapter 4

An Analysis of Collocation on GPUs for Deep Learning Training

Deep learning training is an expensive process that extensively uses GPUs. However, not all model training saturates modern powerful GPUs. To create guidelines for such cases, this paper examines the performance of the different collocation methods available on NVIDIA GPUs: *naïvely* submitting multiple processes on the same GPU using multiple streams, utilizing *Multi-Process Service (MPS)*, and enabling the *Multi-Instance GPU (MIG)*. Our results demonstrate that collocating multiple model training runs yields significant benefits, leading to up to three times training throughput despite increased epoch time. On the other hand, the aggregate memory footprint and compute needs of the models trained in parallel must fit the available memory and compute resources of the GPU. MIG can be beneficial thanks to its interference-free partitioning but can suffer from sub-optimal GPU utilization with dynamic or mixed workloads. In general, we recommend MPS as the best-performing and most flexible form of collocation for a single user submitting training jobs.

4.1 Introduction

Today's GPUs are significantly more powerful than those of a decade ago. Modern GPUs, together with larger datasets, facilitate the exponential growth of deep learning models. Many data scientists, however, do not require large models in practice. For example, a problem may not have a large enough dataset to warrant a large model[1], or the ideal batch size for training the model may not be large enough to utilize all of the GPU resources [82, 86, 93, 92]. This poses an hardware under-utilization issue [125, 92] when training neural networks as the training process usually takes exclusive access to a GPU. This problem gets exacerbated with each new GPU generation offering more hardware resources.

Workload collocation is a method for increasing hardware utilization by running multiple applications at the same time over the same hardware resources. That way, the device and its resources are shared among the collocated applications. While workload collocation is heavily studied for CPUs [99, 126, 127], its opportunities

[1] Data scientists in our lab routinely use less than half of the requested GPU resources during their model parameter exploration.

and challenges have been largely unexplored for modern GPUs. In addition, unlike CPUs, GPUs lack sophisticated resource-sharing methods such as virtual memory and fine-grained sharing.

Today, there are several methods for workload collocation on a GPU. Firstly, multiple processes can be assigned to the same GPU simultaneously without any explicit process management. Alternatively, the collocation can be more precisely managed, for example via NVIDIA's *Multi-Process Service (MPS)*. Finally, the latest generations of NVIDIA GPUs can be partitioned into fully isolated GPU instances at the hardware level via *Multi-Instance GPU (MIG)*.

This paper analyzes different ways of collocating deep learning model training on NVIDIA GPUs. Specifically, we investigate the strengths and limitations of the new MIG technology in contrast to the older methods. We characterize the performance of the above-mentioned collocation methods on an A100 GPU. We diversify our workload by considering three datasets (ImageNet, ImageNet64x64, Cifar10) representing different sizes (large, medium, small). Furthermore, we acknowledge that the current deep learning landscape employs a wide variety of model architectures. We investigate two popular convolutional models (ResNet, EfficientNetv2) and one transformer model (CaiT). Additionally, we collocate a recommender model with a vision model to demonstrate the merits of workloads containing models that stress different parts of the hardware. Our results highlight that:

- When model training is unable to utilize the full GPU on its own, i.e., when running on our small- and medium-sized training cases or cases that stress different parts of the GPU, training multiple models in collocated fashion presents considerable benefits. On the other hand, for large model training, collocation provides either limited improvements to throughput as the GPU becomes over-saturated or cause model training to crash when the available GPU memory is not big enough to hold the combined memory footprint of the collocated models.

- On all the combinations we evaluated, MPS performs better than naïve and MIG collocation, allowing single-user workloads to get the most out of the hardware with minimal setup required.

- MIG offers strict separation of the GPU's memory and compute resources across the collocated workloads, eliminating interference. It also allows multi-user collocation, unlike MPS, and can achieve higher energy efficiency when the partitions are set well. On the other hand, MIG requires creating hardware partitions a priori. For the cases of well-defined workloads, one can create the ideal MIG partitions and leverage MIG-based collocation. However, for more dynamic workloads where the workload mix changes over time, MIG would require re-partitioning to perform well, whereas other collocation methods still provide benefits.

TABLE 4.1: Models & Datasets

Model	Dataset	#Parameters	Size
ResNet26	Cifar10	17M	small
ResNet50	ImageNet64	24M	medium
ResNet152	ImageNet	59M	large
EfficientNet_v2_s	ImageNet64	22M	medium
CaiT_xxs24_224	ImageNet	12M	large
DLRM	Criteo Terabyte	24B	very large

4.2 Background

This section first provides background on different methods of collocation. Then, we survey related work on workload collocation for deep learning.

4.2.1 Collocation on GPUs

A ***CUDA stream*** [128] is a sequence of operations that execute on the GPU (i.e., kernels and data transfers) in the order they are issued. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can run concurrently. This concurrency helps with overlapping the stall time due to the data transfers between the host CPU and GPU in one stream with work from another stream. We call this type of workload collocation the ***naïve*** method since it offers a limited way for sharing GPU resources. This is because the streams have to share the GPU compute resources in a time-based manner rather than having resources explicitly dedicated for each stream.

The ***multi-process service (MPS)*** [38] enables the host CPU to launch multiple processes on a single GPU. Similar to naïve collocation, these processes share the GPU memory and memory bandwidth. However, unlike naïve collocation, the streaming multiprocessors (SMs) of the GPU are split across the different processes. Assignment of the SMs is done by the MPS daemon automatically, unless explicitly stated by the user, based on the provisioning of the GPU resources needed for each process. This reduces interference across the different processes compared to the naïve approach. One limitation of MPS is that it cannot collocate applications launched by different user accounts for security reasons.

Multi-instance GPU (MIG) [129] is the most recent collocation technology introduced with NVIDIA's Ampere GPUs. It provides hardware support for splitting a GPU into smaller GPU instances. Each instance can run a different process allowing these processes to run in parallel on the same GPU.

An A100 GPU with 40GB memory supports several available partitioning profiles (see 2.2). The smallest possible GPU instance is one with just one memory slice and one compute slice, `1g.5gb`, with 14 streaming multiprocessors (SMs) and 5GB of memory. Consecutively, a `2g.10gb` profile consists of two compute slices (28 SMs) and two memory slices (10 GB of memory). The other available profiles are `3g.20gb`, `4g.20gb`, and `7g.40gb`. The last profile consists of almost all of the

GPU resources. However, using the GPU without MIG mode is not analogous to running this large profile as the compute capability of the GPU is hampered slightly due to MIG management overhead; i.e. the reduced compute slice as mentioned above (10 SMs). Each partition is strictly separated in terms of hardware resources preventing any form of interference across partitions.

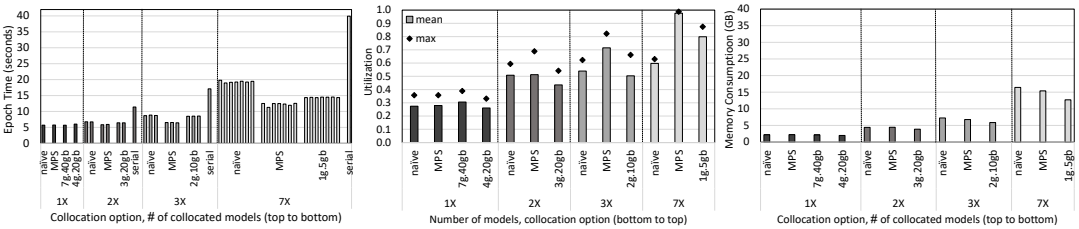
Many different partitions are possible as long as the maximum resource capacity is not exceeded. The partitioning rules are set by the GPU itself, and the allowed set of instances and configurations varies across different types of NVIDIA GPUs (A100, A30, H100, H200). Finally, a GPU instance may also be split into multiple compute instances from the compute side with unified memory. This can be useful when compute and memory requirements do not follow the same pattern. For example, one could run a memory intensive model and a compute intensive model with isolated compute instances on a single GPU instance.

4.2.2 Related work

Collocation on GPUs have been studied in two dimensions: software and hardware approaches. Software approaches either focus on developing better primitives for collocation on GPUs or provisioning the resources of GPUs for running multiple applications [130, 131, 61]. In contrast, hardware approaches propose micro-architectural changes to GPUs to enable finer-grained and more precise multi-application execution within a GPU considering performance, utilization, and quality of service trade-offs [132, 133, 134, 135, 136, 137].

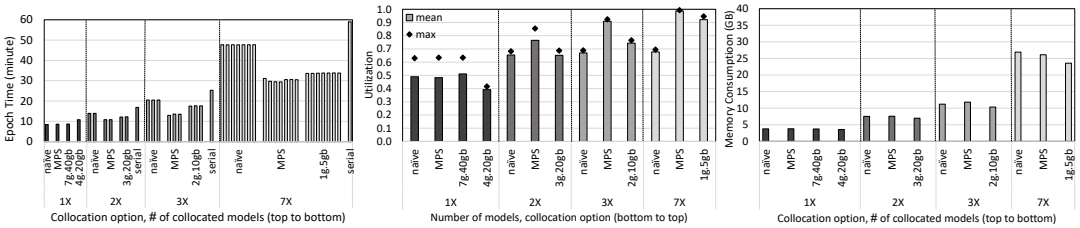
MIG is a relatively new technology and there have not been many works that thoroughly explore its possibilities. HFTA [93] is a mechanism to fuse multiple model training runs for hyper-parameter tuning into one training run. The authors show the effectiveness of HFTA compared to using MPS or MIG to run multiple training runs in parallel. MISO [138] runs MPS on a `7g.40gb` MIG instance to predict the best MIG configuration for different jobs. Finally, Li et al. [139] characterize performance of only MIG using deep learning models focusing on time and energy metrics.

In general, our work is orthogonal to these works since we investigate the strengths and limitations of MIG in contrast to the older collocation techniques such as MPS and naïve collocation and use workloads of different sizes.



(A) Epoch time (B) GPU utilization (C) Memory footprint

FIGURE 4.1: Small: ResNet26 + Cifar10 (batch size = 128).



(A) Epoch time (B) GPU utilization (C) Memory footprint

FIGURE 4.2: Medium: EfficientNet_S + ImageNet64 (batch size = 128).

4.3 Impact of Collocation

4.3.1 Setup & Methodology

System. Our experiments run on a DGX Station A100, composed of an AMD EPYC 7742 CPU (64 cores, 512GB RAM) and four A100 40GB GPUs (108 SMs). Each of the A100 GPUs have 40GB of VRAM and support up to 7 MIG instances with at least 5 GB of memory per instance (see 4.2.1).

Experiments. The experiments are devised with varying dataset sizes [140, 141, 1, 142] and models [143, 144, 145, 146] to assess the performance of collocating deep learning training under different loads (4.1). We orchestrate the execution of the workloads via a benchmarking framework [147]. The vision models are sourced from the TIMM library [113], the recommender model from Facebook Research [146], and we are using the latest version of PyTorch as of the start of our experiments (2.0) [24].

4.3.2 Uniform Collocation

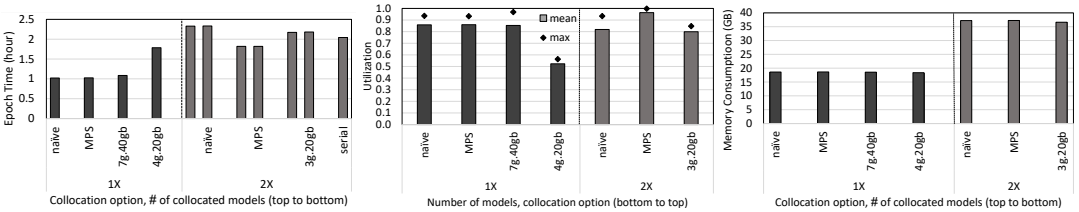
Figures 4.1-4.3 illustrate the results of our uniform collocation experiments. Each figure illustrates a particular model and dataset combination (as subset of the listed combinations in 4.1).[2] Bars that are grouped together form one collocated workload with models trained in parallel. The different degrees of collocation are separated by dotted vertical lines. The four non-collocated cases, which do not run any models in parallel, are the first four bars and form our baselines.

Time per Epoch

Our main performance metric when comparing the effectiveness of different collocation methods is ***Time per epoch***. We time the second epoch of training, skipping the first one as warm-up.

Looking at the first four bars of Figures 4.1a-4.3a, reveals that there is a little variation between the first three non-collocated workloads: `naïve`, `mps`, and `7g.40gb`. This indicates that MPS and MIG have negligible overhead. On the other hand, we see the impact of having fewer resources available on the `4g.20gb` MIG instance as the workloads get larger in Figures 4.2a-4.3a.

[2] A larger set of results can be found in our longer report [17].



(A) Epoch time (B) GPU utilization (C) Memory footprint

FIGURE 4.3: Large: CaiT + ImageNet (batch size = 128).

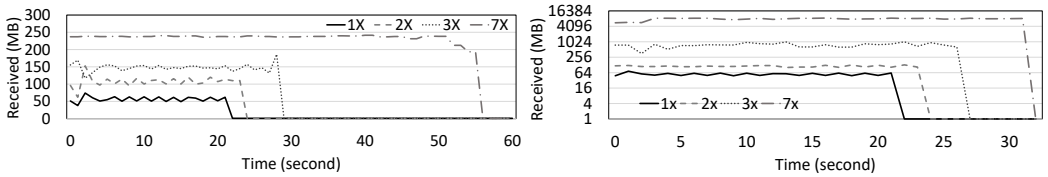
Going over to the collocated runs, comparing across the different collocation mechanisms on Figures 4.1a-4.3a reveals that MIG-based collocation performs better as the degree of parallelism increases (especially to 7). MPS reveals itself as a clear winner, offering the best performance across the board. In contrast, naïve collocation is the least effective. We attribute the superior performance of MPS to its more flexible resource management allowing more effective collocation (as 4.3.4 also shows) and the lower compute resources that are available to MIG (4.2.1).

As expected, collocation impacts the time it takes to train the individual models due to interference across the collocated runs. Additionally, as the degree of collocation increases, so does the total time to train the models. On the other hand, multiple models finish training simultaneously, increasing training throughput. For example, except for the large workloads, 2-way collocation delivers two models in roughly the same time as no-collocation delivers one model. 3-way collocation with MPS and MIG leads to a 50-110% increase in time per epoch compared to non-collocated case while delivering three model training runs instead of one. 7-way collocation with MPS and MIG only increases the runtime 2X-2.5X for our smallest workload (4.1) while delivering 7 models in parallel. These results clearly show that collocation is valuable when a single training run is not large enough for the available GPU compute and memory resources; e.g., the *small* and *medium* cases.

However, the picture shifts considerably with the large workloads (4.3). We no longer see improvements for all of the collocated runs. MPS remains strong and is the only form of collocation that remains beneficial in terms of throughput. Under naïve collocation, one epoch of training takes roughly as long as training the models in sequence without collocation. MIG fairs a little better under 2-way collocation, but is not advantageous. Additionally, 3-way and 7-way collocation becomes impossible due to memory constraints.

GPU utilization

We use SM Activity to track ***GPU utilization***, [16], which is reported by the `dcgm` tool [148]. For the small case and 7-way collocation, the benefits of collocation become very visible. With ResNet's embarrassingly parallel nature and the larger batch size allowing even more parallelism, high utilization of the GPU compute resources is achieved without overloading the GPU (4.1b). The medium case reflects the same pattern, though starts hitting compute resource boundaries under 7-way collocation, as seen in 4.2b. As a result, collocation provides considerable benefits



(A) Naïve (B) Multi-Process Service

FIGURE 4.4: Traffic from CPU to GPU during the second epoch of ResNet26 + Cifar10 (batch size 32) training.

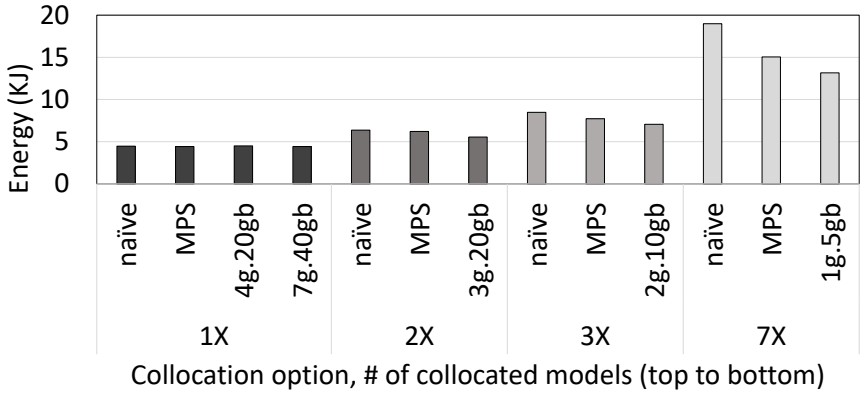


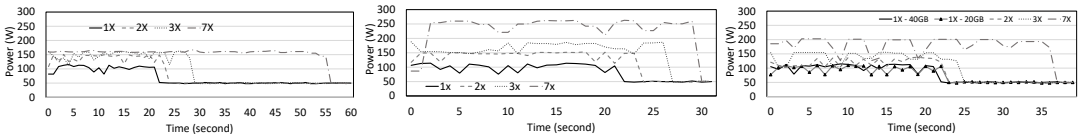
FIGURE 4.5: GPU energy consumption to complete the 2nd epoch of ResNet26 + Cifar10 (batch size 32) training.

for the small and medium cases with MIG and especially with MPS. For the large case (4.3b), there is little variety in the GPU utilization across different cases.

GPU memory footprint

Finally, Figures 4.1c-4.3c report the aggregate ***memory footprint*** on the GPU for different collocation methods for each workload. We use `nvidia-smi` to collect the memory consumption for the whole GPU after a full epoch of training to signify how much memory is needed for the model to train. The figures demonstrate that the increase in memory footprint with collocation is proportional to the degree of collocation. This is an expected result as the models are not sharing data across collocated runs in these experiments.

Notably, MIG collocation shows slightly smaller memory footprints than the two other options, which prompted us to delve deeper into PyTorch's memory allocation. We discovered that PyTorch adjusts the memory footprint depending on the total available memory, which is less in the case of non-`7g.40gb` MIG instances compared to whole GPU memory available under MPS and naïve. Switching the memory allocator backend from PyTorch's native implementation to CUDA's built-in asynchronous allocator removes the differences in the memory footprint of different collocation methods. However, we do not recommend this switch as it slows down the training process.



(A) Naïve (B) Multi-Process Service (C) Multi-Instance GPU

FIGURE 4.6: GPU power usage during the second epoch of ResNet26 + Cifar10 (batch size 32) training.

Interconnect Traffic

4.4 reports the amount of bytes received over time by the GPU measured by `dcgm`'s `pcie_rx_bytes`. We compare naïve and MPS collocation during the second epoch of small ResNet training with batch size 32. We pick this small case as it benefits greatly from collocation and can highlight the differences across the collocation scenarios more effectively. MIG is omitted here due to `dcgm` not providing the readings for this metric under MIG as a result of the GPU being split into multiple instances.

For lower degrees of collocation, naïve collocation leads to a linear increase in data transferred over PCIe from CPU to GPU with respect to degree of collocation. On the other hand, for the 7-way case, there is less work being done per unit of time for each training run leading to sub-linear PCIe traffic. This is likely caused by the throughput benefits of collocation taking a huge hit under naïve collocation, as shown in 4.3.2. In contrast, MPS exhibits a super-linear increase in PCIe utilization when collocating models. In addition to the data transfers for the collocated runs, MPS increases the kernel launch processes since it is able to eliminate false dependencies and share the GPU resources more effectively across the collocated kernels (4.2.1).

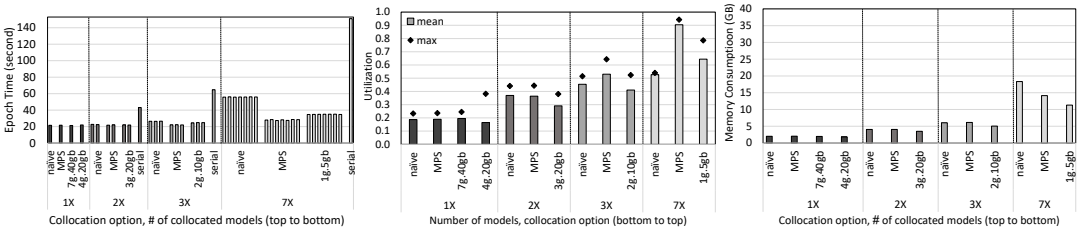
Energy Consumption

Finally, we look at the power usage and GPU energy consumption using `dcgm`'s `power_usage` (watts) and `total_energy_consumption` (joules), respectively, for the small ResNet training. Figure 4.6 shows that collocation scenarios that are highly effective may run on higher power but finish much quicker. This is due to higher GPU utilization under MPS and MIG. MIG exhibits significantly lower wattage under 7-way collocation than MPS while training slightly slower. The benefits of this can be seen in Figure 4.5, which reports the total GPU energy consumption of the second epoch of the model training. While requiring higher power usage per unit of time, MPS spends less energy compared to naïve collocation thanks to finishing training faster. While not as fast as MPS, MIG in general exhibits the lowest GPU energy footprint.

4.3.3 Additional Uniform Collocation Results

As part of our investigation of the collocation mechanisms, we have also experimented with varying the batch size and tested out additional model and dataset

combinations. We are sharing the results from those experiments in this subsection for completeness in Figures 4.7-4.11, even though they do not change the key conclusions of this chapter.



(A) Epoch time (B) GPU utilization (C) Memory footprint

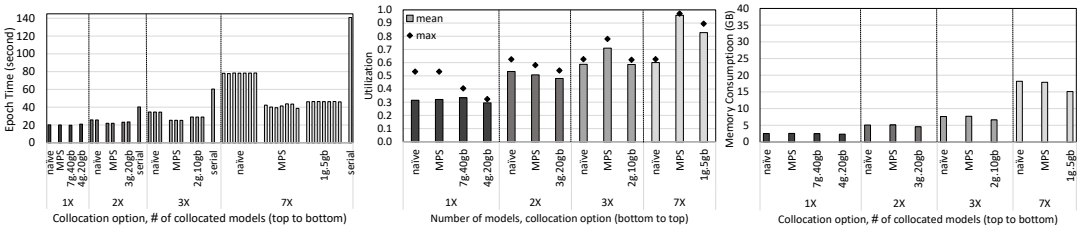
FIGURE 4.7: Small: ResNet26 + Cifar10 (batch size = 32).

4.3.4 Mixed Workloads

So far, we focused on homogeneous collocation scenarios. Such cases can be extremely useful in practice when a data scientist is performing hyper-parameter tuning to come up with the ideal set of parameters for a model repeatedly running the same model with a different set of parameters. On the other hand, there is also value in investigating non-homogeneous collocation scenarios to observe what happens when individual training runs stress the GPU unequally.

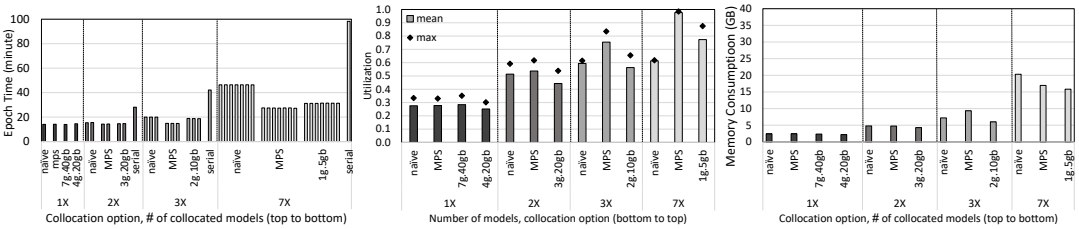
We select combinations of small, medium, and large ResNet models with corresponding dataset sizes to collocate heterogeneously (as listed in Table 4.1). For the MIG workloads, these run on `1g.5gb`, `2g.10gb` and `4g.20gb`, respectively. We opted to keep a static MIG configuration in this experiment since in a real-world scenario, e.g., in a data center, the MIG partitions would already be set and reallocating resources after each training run could be impractical.

Figure 4.12 details the total execution time for training the collocated models using the different collocation methods in comparison to training them back to back, *serial*, without collocation. We see that the benefits of collocation vary heavily across workloads. For larger workloads such as "S+M+M" and "S+S+M+M", naïve and MPS collocation provide sizeable benefits by training the small model without impacting the medium one. In general, the flexibility of both naïve collocation and MPS is a great advantage here over MIG.



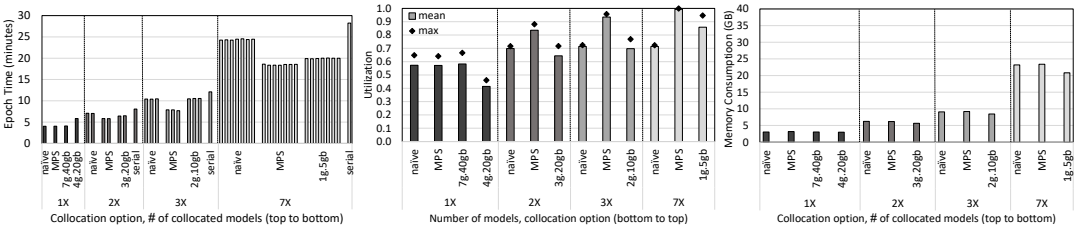
(A) Epoch time (B) GPU utilization (C) Memory footprint

FIGURE 4.8: Small: EfficientNet_s + Cifar10 (batch size = 128).



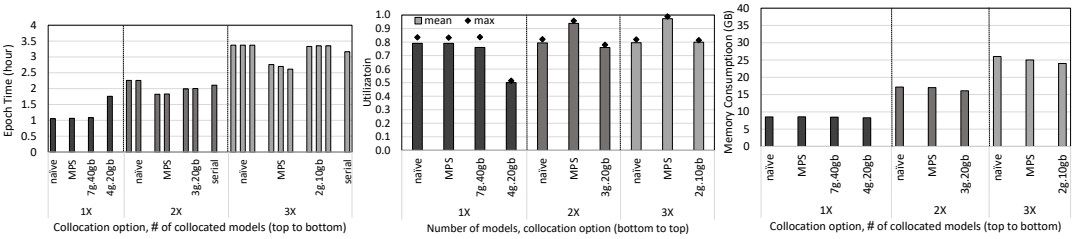
(A) Epoch time (B) GPU utilization (C) Memory footprint

FIGURE 4.9: Medium: ResNet50 + ImageNet64 (batch size = 32).



(A) Epoch time (B) GPU utilization (C) Memory footprint

FIGURE 4.10: Medium: ResNet50 + ImageNet64 (batch size = 128).



(A) Epoch time (B) GPU utilization (C) Memory footprint

FIGURE 4.11: Large: ResNet152 + ImageNet (batch size = 32).

4.13 dives deeper into the "S+M+M+M" workload to observe how the GPU utilization and memory footprint changes over time during collocated runs with naïve, MPS, and MIG collocation. We pick this mix as it is the one that utilizes MIG instances the best. The GPU utilization under MIG gets lowered after the small model finishes, since MIG is unable to fill-up the corresponding instance with more work. On the other hand, naïve and MPS are able to keep similar GPU utilization throughout. In contrast, the memory footprint follows a similar trend for all collocation strategies. It is higher in the beginning as all four models are training. The values then drop off quickly once the small model finishes.

Furthermore, to investigate the impact of collocating mixed workloads that stress different hardware resources, we show the results of collocating a recommender model with a large vision model training in Table 4.2. We configure two `3g` MIG compute instances to share memory as the recommender model does not fit into the memory of smaller GPU instances.

Adding a memory-heavy model such as the recommender greatly promotes collocation. While the training time for individual runs increase slightly, the total time to

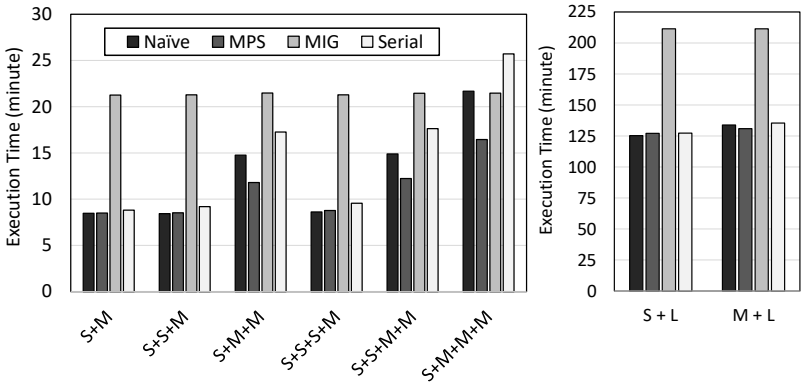
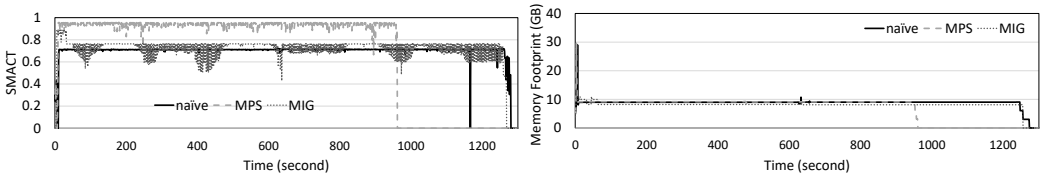


FIGURE 4.12: Time for training mixed vision workloads with & without (serial) collocation for two epochs.



(A) Streaming Multiprocessor Activity (SMACT) (B) Memory footprint

FIGURE 4.13: GPU utilization and memory footprint over time for S+M+M+M from 4.12.

finish the whole workload gets reduced by 5-10%. Furthermore, one can collocate more compute-intensive models such as ResNet152 together with the Recommender model after the first ResNet training completes. As before, memory consumption roughly corresponds to the sum of both models. However, GPU utilization does not increase. Under MIG, unfortunately, only part of the computing power of the GPU can be assigned to ResNet, even though the recommender requires little.

4.3.5 Summary & Collocation Guidelines

Based on the results we covered, we now provide some guidelines for deep learning training collocation.

- Workload collocation is highly beneficial when the aggregate compute and memory needs of the collocated deep learning training runs fit the GPU (*SMACT <= 80%*).
- Collocation gives diminishing returns when the GPU utilization of an individual run is already close to 100%.
- The aggregate memory footprint of the collocated runs can effectively be estimated by the sum of the memory footprints of the individual runs and cannot exceed the available memory on the GPU.
- MPS achieves better performance across the board thanks to its flexible distribution of hardware resources among the collocated runs. On the other hand, it requires higher interconnect bandwidth.

TABLE 4.2: Mixed collocation of memory-intensive recommender and compute-intensive vision models. Recommender time is for one training block plus validation. ResNet time is for one epoch. The reported decrease in total time (%) is relative to the sequential run.

Collocation	Time (h)			GPU Util.	Memory (GB)
	Recom.	ResNet	**Total**		
Recom. (no-colloc)	5.36	-	**6.41**	5%	29.14
ResNet152 (no-colloc)	-	1.05		82%	8.47
Naïve	6.09	1.11	**6.09 (-5%)**	81%	37.75
MPS	5.57	1.10	**5.57 (-13%)**	81%	37.62
MIG (shared)	5.60	1.40	**5.60 (-13%)**	39%	37.86

- MIG can support collocation effectively when a strict separation is required among the runs thanks to its rigid partitioning even though this partitioning leads to sub-optimal performance compared to MPS. Furthermore, MIG exhibits higher energy efficiency on GPUs when the instances are configured well for the workload.

4.4 Conclusion

In this paper, we provide a performance characterization on a modern GPU device that has support for multiple means of GPU collocation: naïve, MPS, and MIG. Our results demonstrate that GPU collocation is highly beneficial for small- and medium-sized workloads that cannot fully saturate the whole GPU. Although per-model training is overall slower, parallel execution of workloads can utilize GPU resources more effectively, increasing training throughput. MIG notably requires a rigid setup while providing full isolation across its instances.

If the workload across the instances is imbalanced, runs that finish early will leave some instances idle, unless there is other work that could be allocated over those instances. Naïve collocation and MPS, on the other hand, can utilize the resources released by the finished work, increasing the training performance of models that require more time to train. In general, MPS provides the best collocation performance, if not the most energy efficient.

In this work, we limited our focus to training on a single GPU since NVIDIA does not allow multi-GPU training with MIG. we limited our focus to training on a single GPU since NVIDIA does not allow multi-GPU training with MIG. In a data center, many workloads can be collocated not only on the same GPU but also on the same server. Therefore, studying the impact of collocation while running other workloads on other GPUs on the same device would be interesting future work. Furthermore, considering the results with the recommender model, further investigations of the shared memory instances of MIG would be worthwhile.

Chapter 5

GPUMemNet: GPU Memory Estimator for Neural Network Training

As deep learning continues to revolutionize various fields, its reliance on high-performance hardware has become increasingly evident. As the backbone of modern deep learning, GPUs provide the computational power necessary to train complex models. However, despite their capabilities, GPUs are often underutilized due to inefficiencies in workload management. Accurately estimating GPU memory usage is crucial not only for optimizing hardware efficiency by increasing batch size but also for facilitating effective task collocation in high-performance computing environments, where resource managers can automatically allocate workloads to maximize utilization.

This chapter introduces GPUMemNet, a novel machine learning-based approach for estimating GPU memory requirements. In addition to presenting the methodology underlying GPUMemNet, this chapter provides access to the dataset used for training the estimator, along with all relevant artifacts of this study. Furthermore, it offers key insights into the GPU memory demands of deep learning training tasks, contributing to a deeper understanding of resource allocation strategies in high-performance computing settings.

5.1 Introduction

Despite GPUs' growing computational capabilities, they remain underutilized in many real-world applications. One primary reason for this underutilization is that deep learning workloads often fail to saturate GPU capacity, combined with the lack of efficient, fine-grained GPU sharing mechanisms comparable to CPU virtual memory. Traditional approaches to scheduling deep learning workloads tend to treat these tasks as "black boxes," assigning entire GPUs without considering the possibility of collocating multiple tasks for improved efficiency. This leads to significant inefficiencies and underutilization when single tasks do not saturate the assigned GPU.

One promising approach to address GPU underutilization is the automatic collocation of multiple training tasks on a single GPU. However, collocation is not without its challenges. The most significant issues are (1) avoiding out-of-memory (OOM) errors that occur when multiple tasks exceed available GPU memory, and (2) managing interference between tasks, which can degrade performance. In this chapter, we tackle the former issue focusing on deep learning training workloads. Efficient and reliable estimation of GPU memory requirements is crucial to overcoming these issues, enabling resource managers to allocate GPU memory effectively while minimizing OOM crashes.

Currently available GPU memory estimation approaches [61, 149, 150] have significant limitations. They drastically over- or under-estimate the memory requirements of deep learning training due to not accounting for GPU memory optimization techniques like memory sharing or reuse at runtime. Making decisions based on overestimated values wastes collocation potential. Conversely, under-estimations might lead to crashes due to OOM errors when we collocate. Recovering from such crashes imposes overhead that penalize what we might gain from collocation. Overall, these inaccuracies make it challenging for resource managers to effectively allocate GPU resources and prevent over-provisioning or crashes.

To address the issue of memory over-/under-estimation in deep learning training, we develop GPUMemNet, a framework composed of dataset building scripts, built dataset cleaning scripts, and models aimed at accurately estimating GPU memory requirements for deep learning models during training, allowing for safe and effective task collocation. Furthermore, to aid with GPUMemnet, we build a comprehensive dataset of training tasks across diverse network architectures, including MLPs, CNNs, and Transformers, all implemented in PyTorch.

This chapter's contributions can be summarized as follows:

- We demonstrate that the state-of-the-art and open-source memory estimation approaches for deep learning training over- or under-estimate (Our experimental observations indicate that both Horus and FakeTensor can significantly misestimate GPU memory requirements, with discrepancies of approximately 350GB and 470GB, respectively, compared to the actual model needs.) the GPU memory needs of the training process emphasizing the need for a more sophisticated by still lightweight memory estimator for deep learning training.

- We show why the memory estimation challenge for deep learning training should be modeled as a classification tasks instead of a regression and provide guidelines for how to collect data for different neural network architectures in order to train accurate memory estimators for deep learning training. We call the overall blueprint for how to collect data and train models for GPU memory usage estimations, ***GPUMemNet***.

- Following GPUMemNet, we build a set of classification models based on multilayer perceptrons (MLP) and Transformers, trained on datasets collected from real deep learning training runs. GPUMemNet maps different training tasks into

memory usage buckets. Based on our evaluation, GPUMemNet can predict the memory needs of the convolutional neural networks (CNNs) and Transformers with 88% and 83% accuracy, respectively, with memory buckets of 8GB increments (0GB-8GB, 8GB-16GB ...), while achieving 97% accuracy, with memory buckets of 1GB increments, for MLPs.

The rest of this chapter is structured as follows. We first delve into the necessary background and motivation in Section 5.2. Section 5.3 provides an overview of related work. In Section 5.4, we describe the dataset construction and analysis. The evaluation of our approach in Section 5.5. Finally, we conclude in Section 5.6.

All artifacts associated with this chapter are available atthe chapter's GitHub page.

5.2 Background & Motivation

The growing complexity of deep learning models has put an increasing demand on computational resources, particularly GPU memory. Modern architectures, including convolutional neural networks (CNNs) and transformers, often require significant memory to store model parameters, intermediate activations, and gradients during training. This makes GPU memory a critical bottleneck in scaling deep learning tasks, especially when working with larger datasets, higher-resolution inputs, or more complex models.

One of the central challenges lies in balancing the trade-offs between memory usage, computational efficiency, and model performance. To address this, researchers and practitioners have explored various techniques to optimize GPU memory usage. Efficient GPU memory utilization is a cornerstone of optimizing deep learning tasks, driven by a variety of innovative techniques. The following subsections highlight some of these techniques and why they make it harder to estimate the exact GPU memory usage for deep learning training.

5.2.1 Memory Optimizations Enabled by Default by the Deep Learning Frameworks

Activation reuse stands out as a crucial method. When activations are no longer required during the backward pass, frameworks like PyTorch [28, 151] and TensorFlow [25] dynamically repurpose their memory for other operations, significantly lowering the memory footprint . However, this approach depends on efficient memory management strategies that vary based on model configurations and layer structures, introducing a level of unpredictability to memory usage.

GPU memory management in popular deep learning frameworks also relies heavily on **dynamic memory allocation**, where tensors, buffers, gradients, and optimizer states are allocated and deallocated during runtime. This dynamic process creates fragmentation—isolated free memory regions that are unusable—complicating memory usage predictions and leading to potential under-utilization or unexpected memory shortages [152, 25, 28].

5.2.2 Memory Optimization Techniques that Must be Enabled by the Users

Layer fusion minimizes intermediate tensor allocations by combining consecutive operations, such as convolution, activation, and batch normalization, into a single kernel.

Gradient checkpointing [153], also known as activation checkpointing, offers a memory-saving solution by selectively storing intermediate activations during the forward pass and recomputing them during backpropagation. This technique is particularly beneficial for large models, as it conserves memory at the expense of increased computational cost. The trade-off between memory savings and computational overhead further complicates the accurate estimation of GPU memory requirements, especially as checkpointed layers and configurations vary across implementations [154].

In addition, **mixed precision** [155] training has revolutionized memory optimization by representing model parameters and activations in float16 instead of float32. This approach significantly reduces memory usage while leveraging NVIDIA Tensor Cores for accelerated operations. However, precision constraints in certain layers necessitate the retention of float32 values to avoid numerical instabilities. These selective adjustments introduce inconsistencies in memory usage and pose challenges for precise memory estimation.

The choice of the **optimizer** for the gradient descent also plays a crucial role in memory consumption. For example, optimizers like Adam [156] and RMSProp [157], which store additional states per parameter, require significantly more memory than solely using stochastic gradient descent (SGD). The specific configurations and parameters chosen directly impact memory usage, further influencing the accuracy of memory predictions.

Beyond the model itself, **asynchronous data loading and preprocessing** can indirectly affect GPU memory usage. Techniques such as data prefetching and augmentation occupy additional memory, leaving less available for model storage. For models with high data throughput, this overhead can become substantial, necessitating careful balancing of resources [158].

Lastly, large models often rely on model parallelism and memory offloading [159] to manage their size. By distributing layers across multiple GPUs or offloading memory to CPUs, these approaches reduce GPU memory bottlenecks but introduce communication overhead. Techniques such as pipeline parallelism, where the model is split into stages across GPUs, add further complexity by requiring memory storage for activations during inter-GPU communication [160, 161, 162, 163].

5.2.3 Impact of Hardware on Memory Allocations

Hardware variability introduces another layer of complexity in memory usage estimations. GPU architecture significantly influences memory allocation and management. For instance, variations between NVIDIA's Volta [164] and Ampere [165] architectures result in differences in memory handling and optimization capabilities, making consistent memory estimation across hardware platforms particularly challenging [166].

5.3 Memory Estimators for Deep Learning

Various approaches have been proposed to estimate memory requirements for deep learning training before the training process starts, including analytical formulas, symbolic computation tools, and integrated libraries within popular deep learning frameworks. However, these methods often fail to capture the nuances of modern GPU memory usage, which is influenced by dynamic optimizations such as the ones listed in Section 5.2, In this section, we review notable techniques for GPU memory estimation: the formula proposed by Yeung et al. [61] for the deep learning task scheduler Horus, the PyTorch's FakeTensor library [149], and the DeepSpeed estimators [150]. Through experimental analysis, we highlight the strengths and shortcomings of these methods, providing insights into their suitability for practical use cases for GPU memory estimation.

Furthermore, Taeho Kim et al. [167] introduce LLMem, **an analytical method** designed to estimate GPU memory consumption during the fine-tuning of large language models (LLMs) across multiple GPUs. LLMem aims to identify the optimal distributed fine-tuning method to prevent GPU out-of-memory (OOM) issues. By analyzing the structure of transformer-based decoder models and the memory usage patterns of various fine-tuning methods, LLMem can predict peak GPU memory usage with minimal error rates. However, LLMem is designed specifically for large language models (LLMs) and fine-tuning scenarios, making it less applicable to other architectures like CNNs or MLPs. While it provides accurate memory estimates, it may not fully account for all variations in model architectures and fine-tuning techniques, limiting its adaptability. Additionally, its performance in dynamic or unforeseen training environments remains untested, which could impact its reliability in real-world applications. While our work is distinct, it can be integrated into the proposed framework and utilized for estimating the specific scenarios where LLMem is effective.

Also, DNNMem [168] employs **an analytical approach** to estimate GPU memory consumption in deep learning models. However, it has several limitations, including the absence of publicly available source code restricting accessibility and community-driven improvements, a focus on single-GPU environments, lack of support for advanced fine-tuning techniques, overlooking optimizer state memory, limited applicability to transformer-based models, framework dependency, and high estimation errors. Analytical methods often face challenges in scalability, as extending them necessitates incorporating additional conditions and variables, leading to more

complex formulas and an increased risk of human error. In contrast, machine learning approaches can effectively address these limitations by expanding the dataset to include new scenarios, thereby enhancing the model's adaptability and accuracy without the need for manually adjusting intricate formulas.

DNNPerf [169] introduces a **machine learning-based** tool designed to predict the runtime performance of deep learning models, focusing on metrics such as GPU memory consumption and training time. DNNPerf represents a model as a directed acyclic computation graph and utilizes a Graph Neural Network (GNN) to incorporate a rich set of performance-related features based on the computational semantics of both nodes and edges. The authors propose an Attention-based Node-Edge Encoder to effectively capture these features. As models increase in complexity, the computational demands of DNNPerf's Graph Neural Network may affect scalability. While DNNPerf shows promise, it has certain limitations. The tool is tailored to specific deep learning frameworks, which may limit its applicability across different platforms or custom implementations. However, it can get extended by adding new data points, which is the benefit of machine-learning mechanisms. Its predictive accuracy may diminish when applied to novel or highly specialized neural network architectures that deviate significantly from the models it was trained on. As deep learning models grow in complexity, the computation required for DNNPerf's GNN-based predictions may become resource-intensive, potentially affecting its scalability and efficiency. The effectiveness of the Attention-based Node-Edge Encoder is contingent on the selected features; omission of relevant features or inclusion of redundant ones could affect prediction accuracy.

DeepSpeed -is a deep learning optimization library from Microsoft that makes distributed training and inference easy, efficient, and effective. - offers a GPU memory estimator that provides insights into memory usage requirements for various configurations, particularly when using ZeRO (Zero Redundancy Optimizer) stages [150]. Users have reported discrepancies between the estimator's predictions and actual memory usage during training. For instance, in certain scenarios, the estimator predicted a per-GPU memory requirement of approximately 17GB, while actual usage reached around 30GB, leading to unexpected out-of-memory (OOM) errors [170]. Furthermore, Complex vision layers, like those with large convolutional kernels, introduce additional memory demands that estimators like DeepSpeed cannot fully capture.

5.3.1 Horus Memory Estimator

Yeung et al. [61] proposes a formula to estimate the expected GPU memory usage of deep learning tasks based on model characteristics and parameters to determine scheduling decisions in the Horus scheduler they also build. This formula provides an estimate for the expected memory usage based on core components in both the forward and backward passes, along with an initialization overhead. More specifically, the expected GPU memory usage Mem_j for a given deep learning job j is calculated through the following:

$$\text{Mem}_j = M_f + M_b + d = (B \times A + P) + (B \times G) + d$$

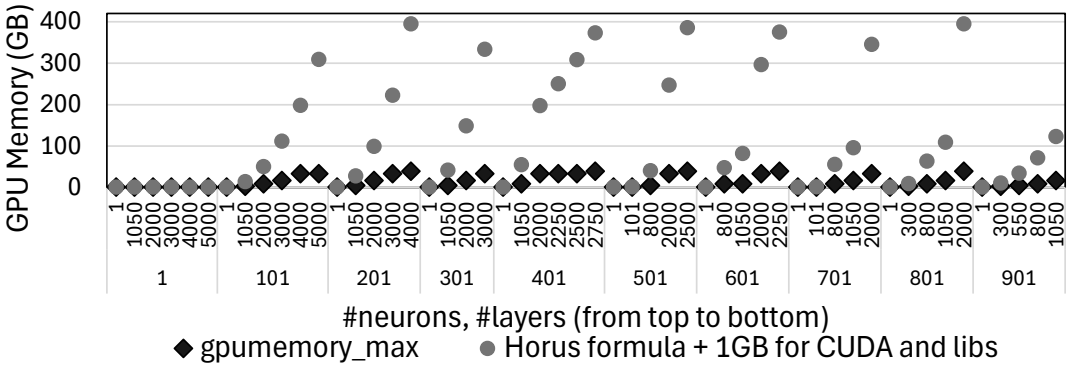


FIGURE 5.1: Comparison of actual GPU memory usage versus Horus-estimated memory requirements for MLP models with varying layer counts and neuron numbers. The discrepancies highlight the limitations of Horus's analytical approach in accurately predicting memory consumption for diverse MLPs.

where

- B is the batch size,
- A is the number of activations,
- G is the number of gradients,
- P is the number of parameters,
- d represents an initialization overhead, and
- M_f and M_b denote the memory requirements for the forward and backward passes, respectively.

Through an experiment, we observe that this formula does not provide accurate predictions for the GPU memory requirements of training even for simple MLP models. The results of our experiment are shown in Figure 5.1. In this experiment, we build a range of MLPs with increasing network depth and width. We train each of them and monitor their GPU memory usage at runtime using `nvidia_smi`. Then, in Figure 5.1, we report the maximum reported value by `nvidia_smi` during training. Also, for each model, we extract its parameters, activations, and batch size to calculate its GPU memory requirement based on the Horus' formula presented above. The results show that this estimation approach can often lead to extremely inaccurate predictions, up to 350GB difference, due to dynamic memory optimizations, such as activation reuse or gradient checkpointing. As a result, this formula is unsuitable for dynamic resource allocation and scheduling decisions for GPUs, especially if one also targets effective collocation of tasks on GPUs.

5.3.2 Memory Estimations with FakeTensor

The primary purpose of the FakeTensor library [149] in PyTorch is to enable symbolic shape propagation and analysis within computational graphs, allowing developers to evaluate model structure, tensor transformations, and layer compatibility without actually allocating memory or performing real computations. FakeTensor achieves this by creating "fake" tensors that maintain essential metadata (like shape and data type) but lack actual data storage, making it valuable for tasks like model compilation, debugging, and compatibility checking across dynamic shapes.

While GPU memory estimation is not FakeTensor's primary function, it can approximate memory requirements based on tensor dimensions and data types, making it helpful for high-level assessments of potential memory usage across network architectures or batch sizes. In "fake mode," FakeTensor simulates tensor allocations, providing a rough estimate of memory demands during forward and backward passes without real allocations. However, these estimates do not incorporate PyTorch's GPU memory optimizations—such as gradient checkpointing, mixed precision, or dynamic memory reuse—which reduce actual memory usage during training.

To investigate the effectiveness of FakeTensor in predicting GPU memory usage, we perform an experiment to compare the actual memory usage of a variety of deep learning models during training on an A100 GPU vs what the FakeTensor estimates for them. The results, which are plotted in Figure 5.2, show that, while estimates are generally within an acceptable range, FakeTensor can sometimes significantly overestimate or underestimate memory usage, given its lack of optimization awareness. These discrepancies, plotted in Figure 5.3, impact resource managers that rely on these estimates, leading to potential underutilization or OOM errors if actual memory requirements deviate from FakeTensor's theoretical predictions.

5.4 GPUMemNet: GPU Memory Estimations using Deep Learning for Deep Learning

In dynamic resource management, particularly within GPU environments, it is imperative to employ models that are both accurate and computationally efficient. Traditional analytical methods, such as those utilized in Horus [61], systematically calculate GPU memory consumption but often lack precision. Conversely, machine learning-based approaches, like Microsoft's graph neural network estimator[169], can offer enhanced accuracy but may introduce significant computational overhead and are not always open-source (e.g., the datasets and/or the estimators), limiting their adaptability. To address these challenges, we propose constructing a systematic and extensible dataset that facilitates continuous and collective improvement. This strategy ensures precise GPU memory estimation while maintaining the efficiency necessary for real-time decision-making in schedulers or resource managers.

In this section, we introduce GPUMemNet, a deep learning-based approach for estimating GPU memory usage for deep learning training tasks. We begin by identifying

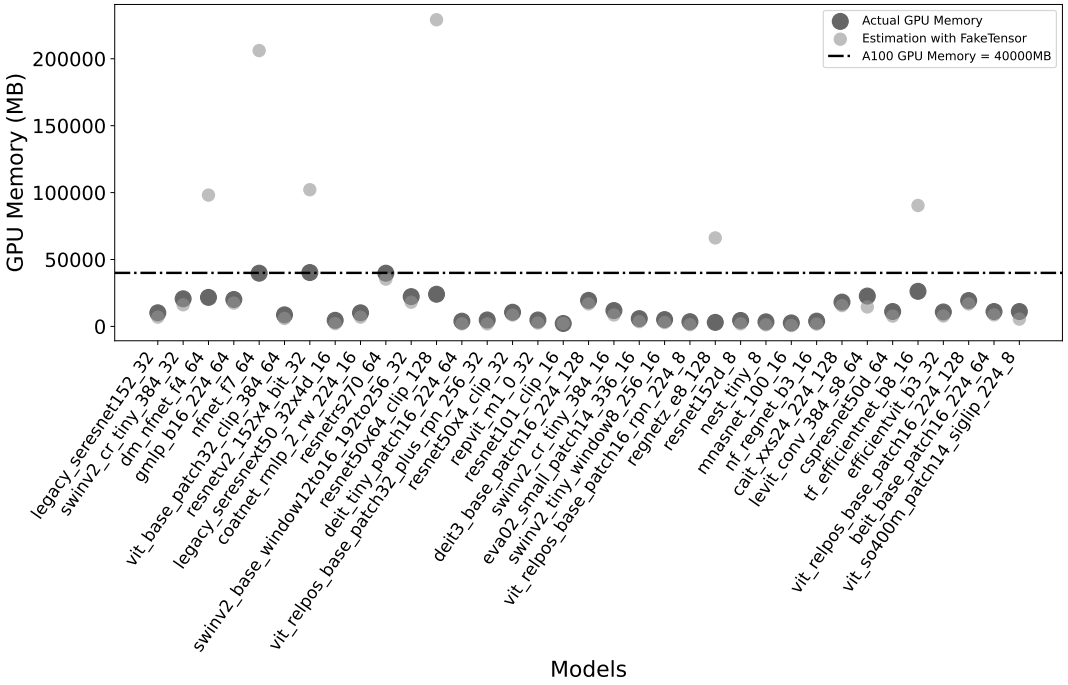


FIGURE 5.2: Actual GPU memory requirement, measured by `nvidia_smi`, and FakeTensor estimation for a range of deep learning models from TIMM library [113] during training.

key neural network characteristics that significantly impact GPU memory consumption. Following this, we investigate the estimator model as a regression task, detailing its challenges. We then describe the construction of the training dataset, ensuring it encompasses a diverse range of network configurations. Additionally, we explore framing the estimator model as a classification task to assess its effectiveness. To validate our approach, we present a proof of concept using Multi-Layer Perceptrons (MLPs), Convolutional Neural Networks (CNNs), and Transformer models.

5.4.1 Neural Network Characteristics that Impact GPU Memory Usage

Number of Layers (Depth): The depth of the network significantly impacts memory usage. For networks with the same batch size and number of parameters, deeper networks (those with more layers) tend to consume less GPU memory than shallower, wider networks.

Activations: In neural networks, activations represent the outputs generated at each layer, which are essential for determining GPU memory usage during training. For MLPs, activations correspond directly to the number of neurons across all layers. Thus, in an MLP with uniform architecture, the total number of activations is simply the sum of neurons in each layer. In convolutional neural networks (CNNs), however, activations refer to the intermediate feature maps produced by convolutional layers, which are way higher than what exist in MLPs.

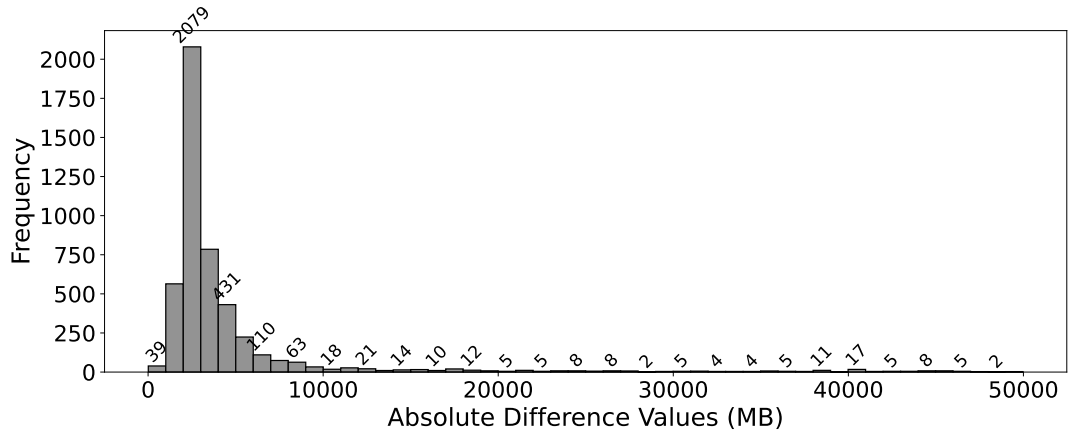


FIGURE 5.3: Histogram of Absolute Value Difference between actual GPU memory requirement of a task and GPU Memory Estimation with FakeTensor Library for Models from TIMM library [113]

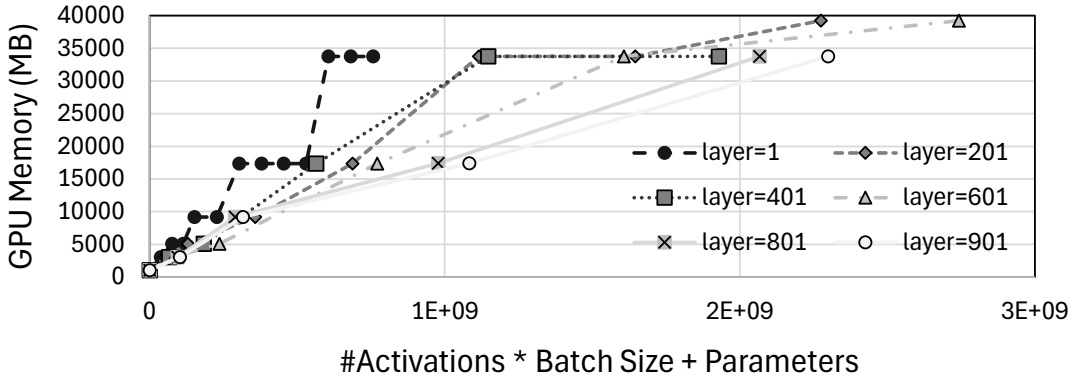


FIGURE 5.4: Staircase growth pattern for memory usage, fully connected networks on ImageNet dataset and with batch_size=32.

Number of Parameters: The weights and biases in the model, which get updated throughout the training process, contribute to memory usage significantly.

Batch Size: Batch size defines the number of training data samples stored on the GPU during each forward and backward pass, which gets multiplied with the number of activations contributing to the GPU memory usage.

Dataset Size and Input Dimensions: Input image dimensions determine the first layer activations and the number of parameters in addition to impacting the training duration. For CNNs, input image size has significant effect on GPU memory requirement as activation in whole network rely on the input image dimension.

To highlight the impact of different network characteristics on GPU memory usage, we initially focus on the fully-connected networks (MLPs) for easier explainability. We construct an initial dataset by running various MLPs on ImageNet [140] using the Keras wrapper for the TensorFlow framework and an A100 GPU. We vary several architectural parameters, including the number of layers, the number of neurons per layer, and batch size. We trained the models on only three different datasets:

MNIST[111], Cifar10[140], and ImageNet [1], and we kept the architecture of all of them to uniform. We record various hardware-related metrics using `nvidia-smi` and `dcgmi` to capture detailed GPU utilization data during model training. While we explain later why this dataset is unsuitable for creating a memory estimator, it still helps us with identifying the memory impact of different network characteristics. Our observations are summarized as follows:

Staircase Growth Pattern: As shown in Figure 5.4, GPU memory usage displays a staircase-like growth pattern, which is attributable to the granularity of GPU memory allocation [171]. GPU memory allocation is done at different granularity levels. Due to the discrete units of GPU memory allocation, different networks may exhibit similar GPU memory usage, even with variations in architecture.

Effect of Network Depth: Figure 5.4 also illustrates that the relationship between network depth and GPU memory usage is complex and depends on specific architectural and optimization factors. Deeper networks can sometimes consume less GPU memory than shallower networks with the same total number of parameters and activations (scaled by batch size) due to advanced memory optimization techniques mentioned in Section 5.2. For instance, memory reuse strategies such as gradient checkpointing can reduce the memory footprint of intermediate activations by recomputing them during the backward pass rather than storing them throughout the forward pass. This allows deeper networks to take advantage of efficient memory usage across layers. In contrast, shallower networks often require all their parameters and activations to be stored simultaneously during training, as there are fewer opportunities for such reuse. However, without these optimizations, deeper networks typically demand more memory because of the greater number of activations that must be retained. Therefore, while deeper networks can have a memory advantage in specific scenarios, this effect is contingent on the application of memory-saving techniques and does not universally apply to all cases.

Batch Size Impact: Figure 5.5 shows that, as expected, larger batch sizes require more GPU memory during training because more activations and intermediate values must be stored to compute gradients and perform parameter updates. In the forward pass, activations are generated for each layer and retained in memory to facilitate gradient computation during the backward pass. With larger batches, the number of activations increases proportionally, as the network processes more data samples simultaneously. Additionally, the backward pass requires storing intermediate values and gradients for each sample in the batch, further increasing memory demands. The gradients for model parameters are computed as the aggregate of contributions from all batch samples, necessitating memory allocation for these intermediate computations. This memory scaling is typically linear with respect to the batch size, meaning that doubling the batch size approximately doubles the memory requirement, assuming the network and input size remain constant.

5.4.2 Estimator Model as a Regression Task

Any resource manager or scheduler that would integrate a model that can estimate the GPU memory usage of deep learning tasks dynamically at runtime, needs to be

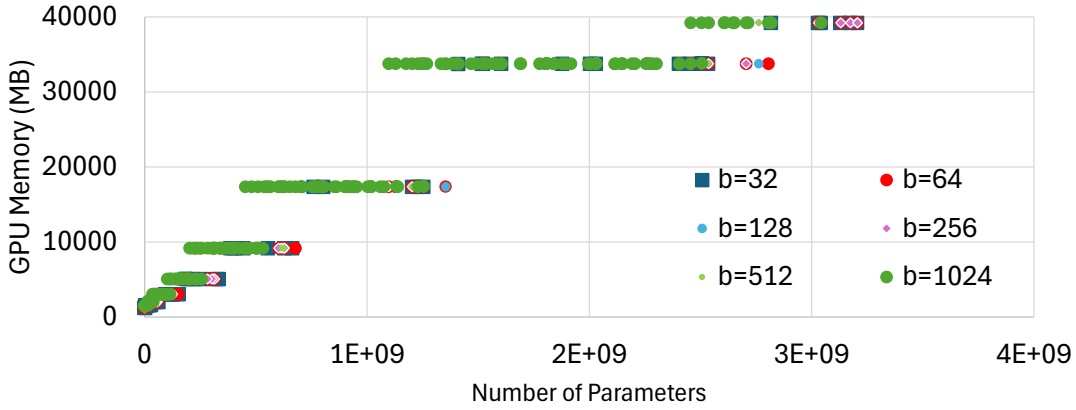


FIGURE 5.5: Batch size effect on different fully connected layers.

lightweight for fast decision making. Therefore, we start by modeling the memory estimation challenge as a regression task aimed at predicting continuous values for the GPU memory usage. There are a variety of lightweight models for regression tasks that does not require a neural-network architecture.

By using the MLP dataset described in Section 5.4.1, we first analyze a variety of regression models using the `lazypredict` [172] Python library. This library allows a systematic way of comparing different models for their accuracy for the task at hand. The input given to the models are `#layers`, `batch_size`, `#parameters`, and `#activations`. The output is the estimated GPU memory usage.

This initial investigation results in the following order for the most accurate models for our task: *ExtraTreeRegressor* [173], *LGBMRegressor* [174], *RandomForestRegressor* [175], and *XGBRegressor* [176]. For example, *ExtraTreeRegressor* manages to predict the memory usage for the MLP dataset with a $+\backslash- \sim 1.2GB$ error margin.

On the other hand, if we perform an analysis of which input features contribute most to the GPU memory usage according to the model, we get the following result for the *ExtraTreeRegressor*: `#layers`=0.0152, `batch_size`=0.0144, `#parameters`=0.699, and `#activations`=0.271. The associated contribution values are from 0 to 1 and the higher the value the higher the contribution. Based on these results, the model thinks that features such as `#layers` and `batch_size` has very little impact on the GPU memory usage. This, we know to be false, based on Figures 5.4 and 5.5. The inconsistencies in feature importance rankings observed in the ExtraTreeRegressor model can be attributed to three key factors. Feature correlation among predictors can distort their relative importance, particularly when highly correlated variables like parameter count and layer count compete for significance [177]. The model's reliance on impurity-based importance metrics presents inherent limitations, especially with high-cardinality or differently scaled features [175]. Additionally, imbalanced feature distributions may lead to underestimation of rare but influential predictors. These considerations are essential for proper interpretation of the model's feature importance assessments [178].

We observe similar anomalies in the way all the top performing regression models

TABLE 5.1: First attempt at modeling as classification with GPU Memory Ranges of 1000MB using the initial MLP dataset for training.

Metric	Accuracy	Precision	Recall	F1-Score
Value	0.8182	0.8205	0.8182	0.8116

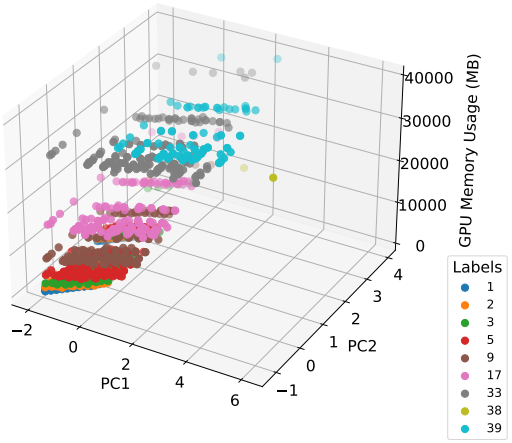
learn about the GPU memory usage of MLPs. This indicates that the models memorize the behavior, overfitting instead of learning. Similarly, using an MLP-based model for this regression task to enable more complex learning does not work. The model does not learn and its Mean Absolute Error (MAE) does not improve over 400 epochs of training. This is likely because of the staircase pattern of memory allocation that we observe in Figure 5.4, which makes the function relating the features to the target variable to be non one-to-one function. The growth in memory use in these models does not fit into a contiguous pattern to be effectively predicted by regression.

Therefore, in our second attempt, in Section 5.4.4, we will approach our task as a classification problem instead of regression. First, to motivate this, we reformulate the challenge as a classification task with 1GB memory range classes. This gives us the high accuracy the results shown in Table 5.1. The reason why the classification task could be more fitting is shown through t-Distributed Stochastic Neighbor Embedding (t-SNE) [179] and Principal Component Analysis (PCA) [180] methods. PCA is a technique that reduces the dimensionality of data by transforming it into a new set of variables, called principal components, which are uncorrelated and ordered by the amount of variance they capture from the original data. This method is particularly useful for simplifying complex datasets while retaining their essential patterns. On the other hand, t-Distributed Stochastic Neighbor Embedding (t-SNE) is a method designed for visualizing high-dimensional data by modeling each high-dimensional object by a two- or three-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability. This approach is especially effective for revealing clusters and intricate structures within the data that may not be apparent through linear methods like PCA. Based on t-SNE and PCA, Figure 5.6 illustrate the distribution of different classes within our dataset. Figure 5.6a highlights the memory range patterns are detectable by the human eye, which explains why a classification task works better for estimating memory. On the other hand, Figure 5.7b shows that the dataset itself does not cover the relevant space, which we delve into next.

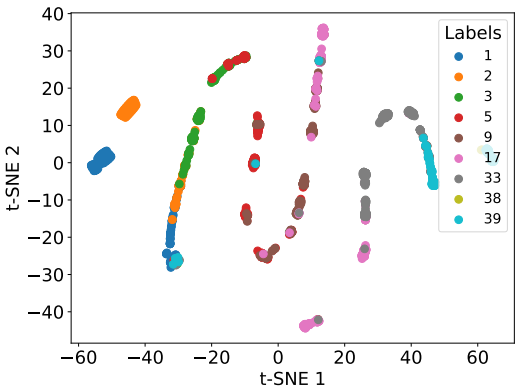
5.4.3 Training Dataset for the Estimator Model

Before delving into the modeling GPU memory usage through classification, we would also like to touch on the approach for preparing a dataset for this task. The initial dataset built using MLPs, as described in Section 5.4.1, is helpful to reason about the key network features that contribute to the memory consumption and identification of the problem type (i.e., regression vs classification). However, it lacks the following key characteristics, which are crucial if one wants to establish a more realistic dataset:

Representativeness of the key input features: The initial MLP dataset represents models with 100s of layers, which helps in analyzing the staircase behavior shown in



(A) The dataset's distribution along the first two principal components and the target variable, GPU Memory Usage to aid in understanding the relationship between the reduced-dimensional features and the target.



(B) t-SNE visualization illustrating data clustering in two-dimensional space. Each number in the legend corresponds to a specific memory usage class within the dataset. This visualization aids in understanding how different classes are distributed in the reduced-dimensional space.

FIGURE 5.6: Analyzing the initial MLP dataset with PCA and t-SNE methods.

Figure 5.4 and why regression-based modeling is unsuitable. However, in realistic scenarios, MLPs are not expected to go beyond 10 layers due to the problem of vanishing gradients. Thus, such data points in the training set for the memory estimator model may lead to misleading learning. One has to focus on realistic ranges for each input feature for the memory estimator while building the training set.

Addition of Key Layers: In real-world models, in addition to the main layers such as fully-connected layers in our initial dataset, there are many other crucial layers such as normalization and dropout layers. The training set must include samples / data points with such layers as well.

Uniform Feature Distribution: To effectively represent each input feature, one has to prepare the dataset where each feature has a uniform distribution. With non-uniform distributions, certain memory values in the range would be missing, hence leading to more errors or over-fitting in model training.

Non-Uniform Layers: MLPs don't always have exactly the same fully-connected layers throughout the model. Some layers may be wider than the others in practice. Same goes for other model architectures. Therefore, in the training dataset, one has to pay attention to having data from models with non-uniform layers across the model

Input and Output Diversity: It may be intuitive to train on a single or a few popular datasets, while building the training set. However, to effectively represent a range of input and output characteristics, one should synthetically adjust input and output size to encompass a wider range of potential training scenarios.

5.4.4 Estimator Model as a Classification Task

Based on the lessons-learned from Sections 5.4.2 and 5.4.3, we now model the GPU memory usage estimation problem as a classification task for deep learning models based on the MLP, CNN, and Transformer architectures. These three architectures already cover a wide range of use cases for popular deep learning applications. For the other network architectures, the overall methodology of collecting the training data for memory estimators would be the same even though the key network parameters for different network architectures change based on the network and real-world usage patterns.

For creating the training datasets, we train the MLP, CNN, and Transformers models with a variety of parameters and setups using PyTorch framework [28], and collect the GPU memory usage data using `nvidia_smi` in addition to capturing the model architecture summaries using the `torchsummary` package.

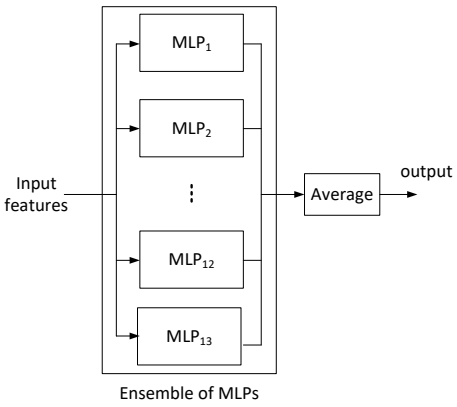
For the memory estimator itself, we train MLP and Transformer architectures using the dataset for each network architecture. The analysis of the sub-dataset with trained model evaluations can be found in their corresponding notebook in the chapter's repository. To enhance the classification performance, we employed an ensemble of 13 multilayer perceptrons (MLPs) with varying depths ranging from 1 to 7 and 1 to 6, incorporating different numbers of neurons. This strategy leverages the diversity among the models to improve generalization and reduce variance, as different architectures capture distinct aspects of the data. Ensemble methods are well-documented to outperform individual models by aggregating their strengths while mitigating their weaknesses [181]. The architectures of the memory estimator models are illustrated in Figure 5.7.

Ensemble methods achieve superior performance at the cost of increased resource demand, as they require concurrent execution of multiple models [182]. The computational burden grows proportionally with the ensemble size, resulting in extended training periods and increased prediction times [183]. Furthermore, the requirement to store parameters for each component model leads to correspondingly higher memory consumption.

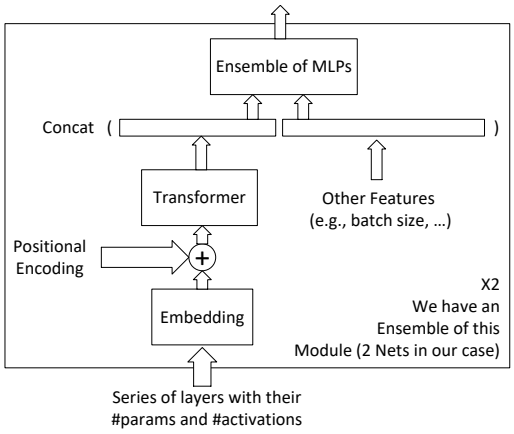
MLPs

MLPs (Multilayer Perceptrons) are often built with several common principles to optimize performance, efficiency, and generalization. Many MLPs follow a pyramid structure, where the number of neurons decreases progressively from the input layer to the output layer to reduce computational complexity and avoid overfitting. Sometimes, architectures may be symmetric, such as in autoencoders [184], where the size of each layer reduces to a bottleneck and then expands again. Initial layers are often wider to capture lower-level features, while deeper layers capture higher-level abstractions. In some cases, practitioners use uniform-width layers to keep the architecture simple and efficient.

For creating our training set for MLPs, we varied model configurations, simulating a range of real-world MLPs commonly used in the deep learning problems. The



(A) MLP Estimator Schematic - Each MLP has different number of layers as well as different number of neurons in the hidden layers



(B) Transformer Estimator Schematic - Ensemble of MLPs have MLPs with different number of layers and neurons in their hidden layers

FIGURE 5.7: Predictor Models (MLP, Transformer) Architecture Schematic

dataset includes results from training MLP-based network architectures with varying input and output sizes and diverse network types, such as uniform, pyramid, and tapered structures. We also considered layer depth, number of neurons per layer, and regularization techniques like dropout [185] and batch normalization [186] to mitigate overfitting. Residual connections, while effective in deeper architectures, were excluded from our MLP dataset as they are uncommon for MLPs compared to CNNs or Transformers.

Table 5.2 summarizes the ranges of the parameters of the MLP sub-dataset while collecting the data points, while we delve into these decisions next.

Input Size: Input sizes range from small (e.g., 4 features for simple tabular data) to large (e.g., 4096 for flattened image-like data). This ensures the study accommodates a variety of real-world MLP applications, from basic feature-based classification to high-dimensional data tasks.

Number of Layers in Modern MLPs:

- **Shallow MLPs (1–3 hidden layers):** Suitable for simpler datasets or problems like tabular data, where additional depth offers diminishing returns.
- **Moderate MLPs (4–10 hidden layers):** Useful for more intricate datasets where additional layers capture complex relationships, such as in image or structured data.
- **Deep MLPs (10–100+ hidden layers):** Excluded from the dataset as these are often better and more accurately handled by specialized architectures like CNNs or Transformers.

Output Size: The chosen range reflects the common design of progressively

TABLE 5.2: Parameter ranges for the MLP dataset.

Parameter	**Range**	**Reasoning**
Input Size	**4** to **4096**	This range captures typical input sizes in MLPs.
Output Size	**1** to **1024**	Aligns with standard design principles for classification and regression tasks
Batch Size	Between **4** and **1024**	Smaller batch sizes support memory-constrained environments, while larger batch sizes optimize GPU utilization and throughput.
FC Layers (BN, Dropout Layers)	- **2** to **12** FC layers - **0** to **11** BN and Dropout layers	Captures the spectrum from shallow to moderately deep networks.
Activation Function	**ReLU**, **ELU**, **Tanh**, etc.	Includes various activation functions used in CNNs.
Architecture Shape	**pyramid**, **uniform**, **bottleneck**, **gradual**	Represents diverse structural patterns.
All Parameters	**27** to **159856482**	The numbers belong to the gathered dataset.
All Activations	**10** to **148066**	The numbers belong to the gathered dataset.
GPU Memory Need	**1443** to **4925** MiB	The numbers belong to the gathered dataset.

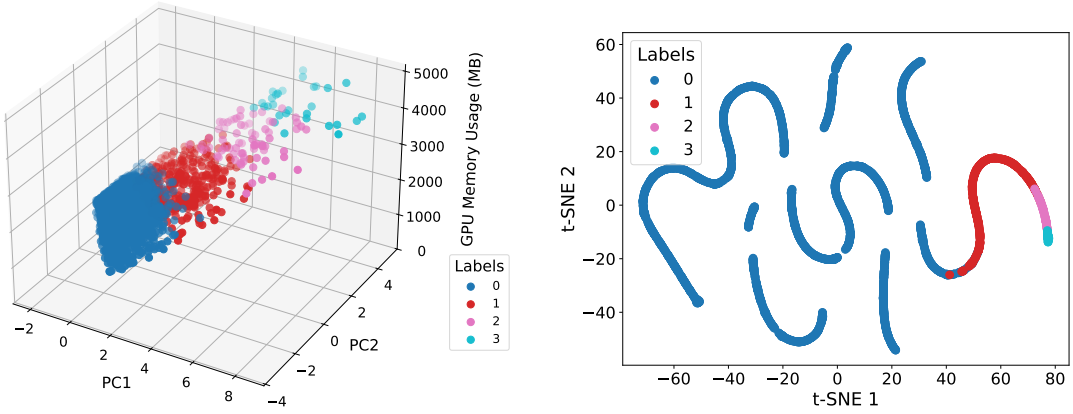
reducing layer sizes to ensure effective information compression and better representation learning. By capping the output size at one-fourth of the input size, the study aligns with established MLP practices in classification and regression tasks.

Batch Size: Batch size is a critical parameter that affects both training performance and GPU memory utilization. Small batch sizes are suitable for constrained environments, while larger batch sizes maximize GPU throughput, supporting diverse use cases and hardware configurations.

Depth: The range of 1 to 10 layers spans from simple to moderately deep MLPs. While deeper networks can model more complex relationships, they introduce challenges such as vanishing gradients, which are addressed by specialized techniques like residual connections or normalization layers.

Architecture Shape:

- **Pyramid:** Mimics the natural structure of many neural networks, where layers progressively shrink.



(A) PCA. (B) t-SNE.

FIGURE 5.8: Analyzing the MLP dataset with PCA and t-SNE similar to the analysis in Figure 5.6.

Dataset	Estimator Model	Accuracy	Precision	Recall	F1 Score
MLP	MLP (1GiB memory ranges)	0.9542	0.9251	0.9542	0.9386
	MLP (2GiB memory ranges)	0.9706	0.9621	0.9706	0.9662
	Transformer (1GiB memory range)	0.9673	0.9599	0.9673	0.9634
	Transformer (2GiB memory range)	0.9804	0.9707	0.9804	0.9755

TABLE 5.3: Accuracy results for the GPU memory usage estimators using MLP and Transformer-based models trained on the MLP dataset.

- **Uniform:** Represents a balanced approach with consistent neurons across layers.
- **Bottleneck:** Enables the learning of compact intermediate representations by drastically reducing neurons in some layers.
- **Gradual:** Offers a smooth reduction in neurons, bridging the extremes of pyramid and uniform structures.

The MLP dataset results in 3000 data points for the memory estimator model to be trained. To collect the dataset, the MLPs are trained on a dummy dataset for a minute. Similar to the initial MLP dataset from Section 5.4.4, we analyzed the dataset using t-SNE and PCA. Figure 5.8 illustrates that the selected features reveal clear patterns, making class distinctions more apparent.

Using the dataset, we train MLP and Transformers models to estimate the GPU memory usage. We partitioned our dataset into 70% for training, 20% for validation, and 10% for testing to ensure robust model evaluation. We perform two types of classification modeling for the MLP dataset, where memory usage ranges are

incremented by 1GiB and 2GiBs. For the MLP-based estimator, the following dataset features are used as input: *number of linear layers*, *number of batch normalization layers*, *number of dropout layers*, *batch size*, *number of parameters*, *number of activations*, and *activation encoding cos/sin*. For the Transformer-based estimator, the following dataset features are used as input: *series of tuples consisting of (layer type, number of activations, number of parameters)* in addition to the *total number of linear layers*, *number of parameters*, *number of activations*, *batch size*, and *number of activations * batch size*. The results for accuracy are presented in Table 5.3. Overall, both models perform well in estimating the memory usage on the collected MLP dataset, while Transformers-based models perform slightly better.

TABLE 5.4: Parameter ranges for the CNN dataset.

Parameter	**Range**	**Reasoning**
Input Channels	**1** to **3**	Covers *grayscale* or *RGB*.
Output Classes	**2** to **22,000**	Covers simple to complex scenarios.
Filters	**16** to **512** with sizes of **3x3, 5x5, 7x7**	Covers common simple to complex scenarios.
Conv2d Layers	**1** to **29**	Covers a range from shallow to deep CNNs.
Activation Function	**ReLU**, **ELU**, **Tanh**, etc.	Includes various activation functions used in CNNs.
Total Activations	**24,514** to **5,317,481,490**	Represents the number of activations across different layers.
Total Parameters	**704** to **329,307,377**	Captures model size variations from lightweight to large CNNs.
Batch Size	**2** to **62**	Ensures flexibility for memory constraints and GPU efficiency.
Conv2d Layers	**1** to **29**	Varies based on depth and complexity of convolutional networks.
BatchNorm2d Layers	**None** to **29**	Accounts for architectures with and without batch normalization.
Dropout Layers	**None** to **29**	Allows varying levels of regularization.
Linear Layers	**1** to **1**	Ensures all CNNs have at least one fully connected layer.
Architecture Shape	**bottleneck**, **uniform**, etc.	Includes a diverse set of CNN structural patterns.
GPU Memory Need	**1,703** to **40,000** MiB	The numbers belong to the gathered dataset.

CNNs

After successfully building the dataset for MLPs to validate the use of deep learning models for learning GPU memory usage patterns, we extended our approach to gather data points for Convolutional Neural Networks (CNNs). Similar to MLPs, we consider several key features that define the real-world CNN model architectures and their impact on the GPU memory requirements. Table 5.4 summarizes the key decisions made for the feature ranges, while we delve into these decisions next.

Input Channels: The number of input channels is randomly set to either 1 (grayscale) or 3 (RGB). These settings reflect common image types. However, this can be extended to include more channels, such as multi-spectral images, which are frequently used in remote sensing and medical imaging applications.

Number of Output Classes: The number of output classes was randomly chosen between 2 and 22,000. This large range was selected to make the network applicable to both simple tasks like binary classification and complex large-scale multi-class problems, such as those in the ImageNet dataset [1] and the Open Image Dataset [187].

Base Number of Filters: The number of filters in the first convolutional layer was chosen randomly between 16 and 512. Filters determine the capacity of a CNN to extract features from an input image. A larger number of filters typically enables the model to learn more complex patterns at the cost of increased memory and computational requirements.

Filter Size: The filter size was selected randomly from 3x3, 5x5, or 7x7. Filter size defines the receptive field of a neuron, which directly affects how much of the spatial context is captured by the network. Smaller filters (e.g., 3x3) are commonly used in modern architectures such as VGGNet [188] due to their efficiency and ability to stack deeply, while larger filters (e.g., 7x7) are sometimes used in the early layers for wider receptive fields [112].

Depth: The depth of the network, defined by the number of convolutional layers, was randomly selected between 1 and 29 layers in the code. This range captures a wide variety of network types, from shallow networks to very deep ones. Deeper architectures can learn more abstract features but may suffer from vanishing gradients and increased memory demands, which are typically mitigated by using techniques like batch normalization and skip connections [186, 112].

Network architecture shapes: The shape of a CNN architecture play a critical role in determining the GPU memory requirement of the model. Seven distinct shapes are considered when gathering the CNN data points based on the real-world use cases.

- **Pyramid:** The pyramid architecture progressively increases the number of filters as the network deepens, following a design philosophy exemplified by VGG [188]. This approach allows deeper layers to capture increasingly complex features, making it suitable for tasks such as image classification, where progressively refined feature extraction is required.

- **Uniform:** In the uniform architecture, the number of filters remains constant across all layers, reducing computational and memory overhead while maintaining simplicity. This architecture is effective in domains where feature complexity remains uniform across layers, such as medical imaging or object detection tasks.

- **Bottleneck:** Inspired by ResNet [112], the bottleneck architecture compresses information by reducing the number of filters in intermediate layers before expanding them again. This design optimizes memory usage and computational cost while preserving representational power, making it ideal for large-scale datasets and real-time applications like ImageNet classification.

- **Gradual:** Gradual architectures employ a linear increase in the number of filters with depth, striking a balance between underfitting and overfitting. This smooth progression is well-suited for tasks requiring nuanced feature extraction, such as facial recognition or fine-grained image classification.

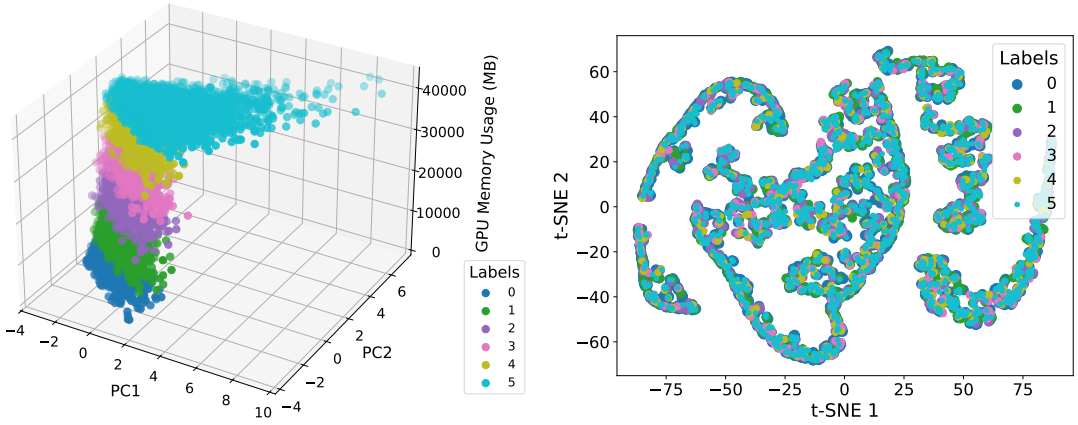
- **Hourglass:** The hourglass design reduces filters towards the middle layers and increases them again in the later layers, mimicking a symmetrical processing flow. This architecture is particularly effective for structured prediction tasks like human pose estimation or semantic segmentation, where features are first compressed and then expanded for fine-grained detail extraction [189].

- **Residual:** Residual architectures incorporate skip connections to facilitate gradient flow during backpropagation, mitigating the vanishing gradient problem [112]. This enables the training of very deep networks (50+ layers) while ensuring stable performance. Residual connections are commonly used in large-scale image classification and segmentation tasks. Noting that when calculating the number of activations for networks with these kind of connections, we should be careful as the number of active activations on GPU memory increases in networks with these connections.

- **Dense:** Inspired by DenseNet [190], dense architectures connect each layer to every other layer in a feed-forward manner, promoting feature reuse and efficient gradient flow. This design reduces the number of parameters and enhances representational power, making it particularly useful for applications requiring precise localization, such as medical imaging and object detection.

To collect the CNN dataset, the CNN models are implemented with these parameter ranges and trained on a dummy dataset for a minute. The Adam optimizer was employed to ensure consistent optimization across experiments. Figure 5.9 shows how the data patterns are for the CNN dataset, which illustrates the harder to learn patterns for the CNNs compared to MLPs (i.e., Figure 5.8), which is also reflected in the memory estimation results presented next.

Using this dataset, we train MLP and Transformers models to estimate the GPU memory usage for CNNs. We partitioned our dataset into 70% for training, 10% for validation, and 20% for testing to ensure robust model evaluation (however in the analysis phase with notebooks, we partitioned our dataset into 70% for training, 20%



(A) PCA. (B) t-SNE.

FIGURE 5.9: Analyzing the CNN dataset with PCA and t-SNE similar to the analysis in Figure 5.6.

Dataset	Estimator Model	Accuracy	Precision	Recall	F1 Score
CNN	MLP (8GB range)	0.8825	0.8816	0.8825	0.8803
	Transformer (8GB range)	0.8449	0.8443	0.8431	0.8431

TABLE 5.5: Accuracy results for the GPU memory usage estimators using MLP and Transformer-based models trained on the CNN dataset.

for validation, and 10% for testing). We set the memory usage ranges to 8GiBs as CNNs in general has higher memory requirements than MLPs, hence the larger ranges. For the MLP-based estimator, the following dataset features are used as input: *depth of the network*, *number of convolutional layers*, *number of batch normalization layers*, *number of dropout layers*, *batch size*, *number of parameters*, *number of activations*, *activation encoding cos/sin*, and *activations * batch size*. For the Transformer-based estimator, the following dataset features are used as input: *series of tuples consisting of (layer type, number of activations, number of parameters)* in addition to all the input features used for the MLP-based model. The results for accuracy are presented in Table 5.5. Overall, while not as accurate as the MLP case, due to the more complex network architecture CNNs have, both models perform well in estimating the memory usage on the collected CNN dataset, while the MLP model perform slightly better.

Transformers

Following the collection of the MLP and CNN datasets, we employ the same systematic approach to collect data for transformer models as well, specifically tailored for real-world natural language processing (NLP) tasks. Table 5.6 summarizes the characteristics of the collected Transformers dataset based on the key decisions made while collecting the data points. We delve into these key decisions next based on the key components of Transformers network architectures.

Number of Classes: The output layer is configured to predict between 2 and 1000

TABLE 5.6: Parameter ranges for the collected Transformers dataset.

Parameter	Range	Reasoning
Linear Layers	**9** to **157**	Covers a range from shallow to deep Transformers.
LayerNorm Count	**4** to **78**	Covers a range from shallow to deep Transformers.
Dropout Layers	**6** to **117**	Covers a range from shallow to deep Transformers.
Total Parameters	**7,292,531** to **3,105,662,679**	Captures model size variations from small to very large-scale Transformers.
Total Activations	**19,184** to **3,194,470**	Represents the number of activations across different layers.
Batch Size	**1** to **128**	Ensures flexibility for memory constraints and GPU efficiency.
GPU Memory Need	**1,683** to **40,369** MiB	Reflects real-world GPU memory consumption for training.

classes, covering both binary and multi-class classification tasks to assess the model's adaptability across various datasets.

Embedding Size: We varied the embedding size from 128 to 2048, reflecting standard practices in transformer architectures. Smaller embeddings suit simpler tasks, while larger embeddings capture complex data patterns.

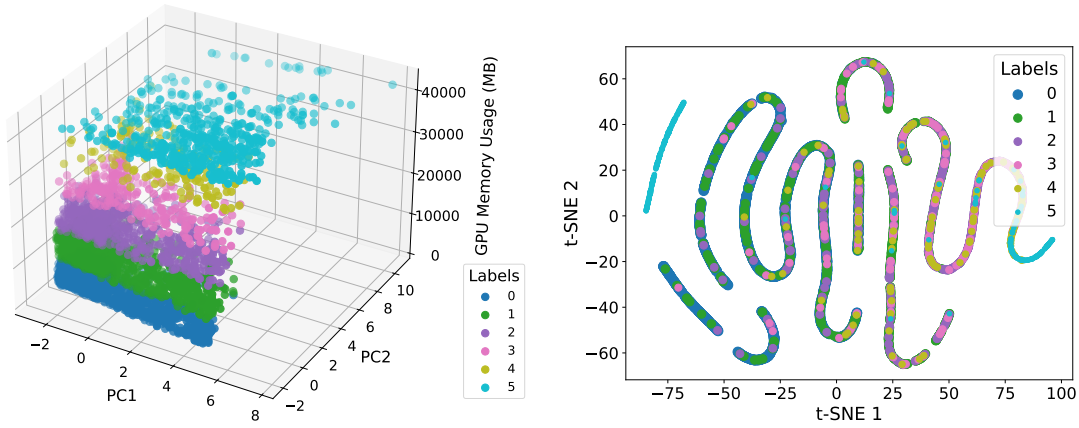
Number of different Layers: The linear, normalization, and dropout layers range from 9, 4, and 6 to 157, 78, and 117 respectively, balancing model depth with computational efficiency. This range allows for effective training across tasks of varying complexity without excessive model depth.

Attention Heads: The number of attention heads varies from 2 to 16, enabling the model to capture diverse relationships within data, thereby enhancing learning performance metric e.g, accuracy.

Feed-Forward Hidden Size: Set between 2 and 4 times the embedding size, this range ensures adequate capacity in the feed-forward layers to process representations from the attention layers effectively.

Dropout Rate: We applied dropout rates from 0.1 to 0.5 to prevent overfitting, in line with best practices that enhance model generalization.

Sequence Length: Fixed at 128, 256, 512, and 1024 tokens, these lengths strike a balance between computational efficiency and the ability to capture long-range dependencies.



(A) PCA. (B) t-SNE.

FIGURE 5.10: Analyzing the Transformer dataset with PCA and t-SNE similar to the analysis in Figure 5.6.

Vocabulary Size: Ranging from 50,000 to 1,000,000, this configuration accommodates a wide variety of NLP datasets while managing memory usage.

Figure 5.10 shows how the data patterns are for the Transformers dataset, which illustrates the classes of memory usage (Figure 5.10a) but even harder to learn patterns (Figure 5.10b) compared to CNNs and MLPs (i.e., Figures 5.9 and 5.8), which is also reflected in the memory estimation results presented next.

Same as the two sub-datasets, we collect the Transformers dataset by training automatically built models on a dummy dataset for a minute. Using this dataset, we train MLP and Transformers models to estimate the GPU memory usage for Transformers. We partitioned our dataset into 70% for training, 10% for validation, and 20% for testing to ensure robust model evaluation (however in the analysis phase with notebooks, we partitioned our dataset into 70% for training, 20% for validation, and 10% for testing). We set the memory usage ranges to 8GiBs. For the MLP-based estimator, the following dataset features are used as input: *depth of the network*, *number of linear layers*, *number of batch normalization layers*, *number of dropout layers*, *batch size*, *number of parameters*, *number of activations*, and *activations * batch size*. For the Transformer-based estimator, the following dataset features are used as input: series of tuples consisting of *(layer type, number of activations, number of parameters)* in addition to all the input features used for the MLP-based model. The results for accuracy are presented in Table 5.7. Overall, similar to CNNs, the results are not as accurate as the MLP case, due to the more complex network architecture Transformers have, but both models perform well in estimating the memory usage on the collected Transformers dataset.

5.5 Evaluation

GPUMemNet models for estimating GPU memory requirements for deep learning training tasks has been trained and tested on the representative but synthetic MLP, CNN, and Transformers network architectures in Section 5.4. Here, we evaluate

Dataset	Estimator	Accuracy	Precision	Recall	F1 Score
Transformers	MLP (8GiB range)	0.8552	0.8524	0.8552	0.8494
	Transformer (8GiB range)	0.8650	0.8650	0.8650	0.8596

TABLE 5.7: Accuracy results for the GPU memory usage estimators using MLP and Transformer-based models trained on the Transformer dataset.

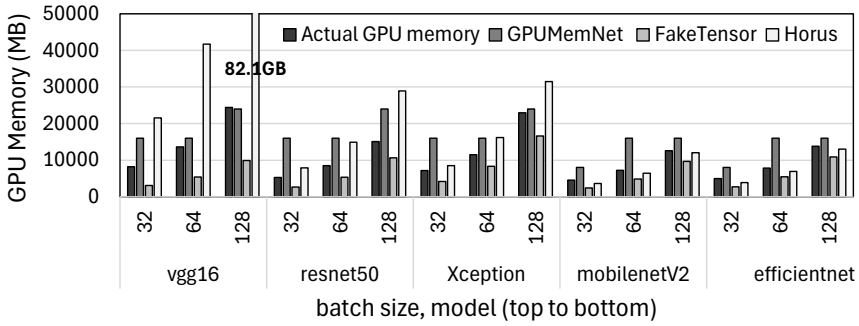


FIGURE 5.11: GPU Memory Estimation for Real-World Unseen CNN Models with Horus, FakeTensor, and GPUMemNet.

the effectiveness of *GPUMemNet* models on deep learning models used in the real world, while comparing its results to the *Horus* (Section 5.3.1) and *FakeTensors* (Section 5.3.2) GPU memory estimators. More specifically, we test these memory estimators on a variety of real CNN and Trasnformers models with varying batch sizes. In the case of *GPUMemNet*, the memory estimator with the higher accuracy is used for the corresponding model architectures; i.e., the MLP model is used from Table 5.5 for estimating the GPU memory requirements of the CNN-based model architectures (checkpointed from its notebook, and added to the test pipeline, which can be found in the chapter's repository). Furthermore, for the transformer-based models, we use MLP from Table 5.7 with 85% accuracy.

5.5.1 Experimental Setup

The experiments are conducted on an NVIDIA DGX Station A100, equipped with a 64-core AMD EPYC 7742 processor, 512 GB of CPU memory, and four NVIDIA A100 GPUs, each featuring 40 GB of high-bandwidth memory (HBM2). The system operates on DGX OS, a customized version of Ubuntu 20.04.4 LTS, with CUDA version 12.2 installed.

For gathering the real GPU memory use, labeled as *Actual GPU memory* in figures, of the deep learning models under test, we train each of them on a synthetic dataset for one minute to make sure that the resource consumption behavior of the models get stable. For sensitivity analysis, we train each model with varying batch sizes as well.

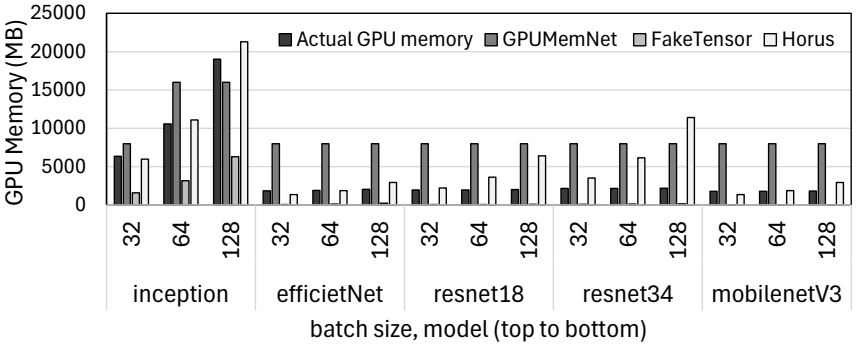


FIGURE 5.12: GPU Memory Estimation for Real-World Unseen CNN Models with Horus, FakeTensor, and GPUMemNet.

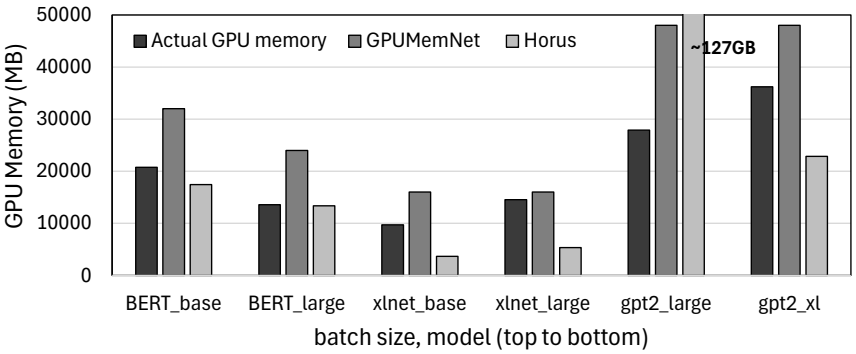


FIGURE 5.13: GPU Memory Estimation for Real-World Unseen Transformers Models with Horus, FakeTensor, and GPUMemNet.

5.5.2 Results

Figures 5.11 and 5.12 and Figure 5.13 present the results for the CNN and Transformers models, respectively. Overall, they demonstrate the higher reliability of the machine learning-based GPU memory estimation using *GPUMemNet* methodology compared to the other state-of-the-art and available GPU memory estimators from the literature.

For the *FakeTensor* library, predicting GPU memory usage for transformer-based models is a compatibility issue. The library does not provide numerical estimates for these models, and therefore is excluded from Figure 5.13. For CNNs, **FakeTensor** tends to underestimate GPU memory usage, with the largest underestimation occurring for *vgg16* with a batch size of 128, where the predicted value falls short by approximately 14.6GB.

Horus' memory estimation formula provides generally acceptable results; however, in certain cases, such as *gpt2_large*, it significantly overestimates GPU memory usage, with a discrepancy of approximately 100GB compared to the actual value.

In contrast, *GPUMemNet* demonstrates superior estimation performance relative to the aforementioned estimators. It only underestimates memory usage in two cases: for *vgg16* with a batch size of 128, where the difference is 0.5GB, and for *inception* with a batch size of 128, where the difference is approximately 3.1GB lower than the actual

value. Compared to *Horus* and *FakeTensors*, these underestimations are significantly low. On the other hand, while still exhibiting a lower mis-estimation compared to the other two methods, *GPUMemNet* overestimates *gpt2_large* by approximately 20GB.

Transformer architectures pose a more significant challenge on memory estimators. This underlines that we need to expand the Transformers datasets for more robust predictions in the future. Specifically, *gpt2_large* is a transformer model composed primarily of self-attention and fully connected layers, and it utilizes `Conv1D` layers with a kernel size of 1 as a functional equivalent of linear layers in the feed-forward network. Since the dataset was built considering only standard `Linear` layers without accounting for such architectural variations, this discrepancy contributes to the observed estimation inaccuracies.

5.6 Conclusion

In this chapter, we introduced GPUMemNet, a machine learning-based framework for estimating GPU memory usage during deep learning model training. Given the increasing demand for GPU resources in deep learning, efficient GPU utilization is critical for improving performance and avoiding costly out-of-memory (OOM) errors. Traditional estimation approaches—such as analytical formulas and tools like FakeTensor—struggle to accurately capture the complexities of modern GPU memory allocation, leading to frequent underestimation or overestimation. Our approach addresses these limitations by leveraging deep learning models trained on a diverse dataset of neural networks.

We systematically construct a dataset covering a wide range of architectures, including MLPs, CNNs, and Transformers, with varying configurations such as depth, batch size, activation functions, normalization layers, and architectural shape patterns (e.g., residual, dense, bottleneck). Through careful data collection and feature engineering, we transformed the GPU memory estimation problem into a classification task, allowing for more robust predictions across different network types.

Our evaluation demonstrate that GPUMemNet achieves high accuracy in predicting GPU memory requirements across various deep learning architectures. The best performance was observed in MLPs, where predictions were within 1GiB accuracy at 97%, while CNNs and Transformers achieved 88% and 86% accuracy within 8GiB, respectively. These results highlight the feasibility of using deep learning to model GPU memory behavior, capturing the nuanced impact of activations, parameters, and architectural choices on memory consumption.

To validate the effectiveness of GPUMemNet in real-world scenarios, we tested it on unseen models from different domains, including standard vision models like ResNet, MobileNet, EfficientNet, Inception, and transformer-based architectures like BERT, XLNet, and GPT-2. Compared to FakeTensor and formula-based estimations (e.g., the Horus formula), GPUMemNet provides more reliable memory predictions. In addition to the GPUMemNet's methodology, we release all the artifacts associated with it, to create a continuous and open framework for robustly estimating GPU

memory requirements of deep learning models in the future. Furthermore, we encourage the community to contribute by extending the dataset to cover more model architectures, and scenarios that the released dataset in this work lacks.

Our results highlight GPUMemNet's potential in reducing OOM errors and inefficient GPU usage when used in resource allocation tools and schedulers for deep learning. By integrating GPUMemNet into a scheduler in the next chapter, Chapter 6, we aim to enhance workload collocation, allowing multiple training tasks to share GPU resources effectively while avoiding OOM failures. Our findings provide a foundation for developing memory-aware scheduling policies, which can optimize performance-energy trade-offs and improve GPU utilization in high-performance computing environments.

Chapter 6

RAD-RM: Resource Manager Collocating Training Tasks on GPUs

The rapid growth of deep learning has driven an unprecedented demand for efficient resource management in computational clusters. With the increasing complexity of models and the volume of tasks executed simultaneously, managing GPU resources effectively has become a critical challenge. GPUs, as the backbone of deep learning workloads, are often underutilized [92, 191] due to many reasons, one of which is inefficient scheduling and resource allocation strategies. While recent proposals for resource managers targeting deep learning [55, 56, 57, 61, 63, 73, 192], address some of these inefficiencies, many fail to explore task collocation on GPUs as a foundational principle for enhancing resource utilization. The few proposals ([193, 194]) that explore collocation ignore the potential out-of-memory crashes.

Chapter 4 highlights that collocation can enhance GPU utilization. However, it is crucial to recognize that GPUs lack virtual memory, meaning workloads exceeding available GPU memory will crash. Collocating multiple workloads, therefore, increases the chances of such crashes due to higher load on the GPU resources. Furthermore, overloading of GPU's compute resources can lead to performance degradation despite achieving higher GPU utilization. To address the former challenge, we introduced GPUMemNet for estimating the memory requirements of deep learning training tasks in Chapter 5 and also reviewed the available methods. To address the latter challenge, one needs to integrate GPU collocation effectively into a resource manager.

In this chapter, we propose a server-level resource manager, ***RAD-RM***, incorporating collocation as a first class citizen through incorporating GPU memory need estimation methods such as GPUMemNet and recovery method against out-of-memory crashes. RAD-RM is build as a framework independent of any machine learning framework. It gets submitted requests as input and determines on which GPUs in the server to schedule them. While RAD-RM targets deep learning training workloads, since it is a software-layer independent of any deep learning codebase, it can be used for other workloads, which use GPUs as well, putting aside the deep-learning-specific GPU memory estimators and solely relying on the recovery method we propose. To the best of our knowledge, RAD-RM is the first resource manager that explicitly handles out-of-memory crashes for task collocation on GPUs.

6.1 Introduction

Despite their critical role in the field of AI, GPUs are often significantly underutilized in real-world scenarios. A well-known study by Microsoft [92] analyzed a two-month-long trace, handling 96,260 jobs submitted by thousands of users, and observed an average GPU utilization of around 52%. A recent work, also from Microsoft [191], examined 400 deep learning jobs, each with an average GPU utilization of 50% or less, identifying 706 low-GPU-utilization issues. Neither study reveals information about the hardware setup, such as the number of GPUs. However, both highlight a persistent issue, demonstrating that GPU utilization in machine learning workloads is frequently far below its potential. This underutilization represents more than just a missed opportunity for performance optimization—it has broader implications for energy efficiency, cost-effectiveness, and environmental sustainability.

When GPUs operate below their potential, the energy consumed to keep them powered is not fully translated into computational productivity. This inefficiency inflates operational costs and contributes to the already substantial carbon footprint of data centers. Moreover, the physical resources invested in designing, manufacturing, and deploying GPUs are only partially leveraged. From raw materials to advanced engineering processes, significant investments are made to create these devices, and underutilization effectively wastes these efforts. In an era where green computing is a growing priority, improving GPU utilization is not merely a technical challenge but a responsibility toward sustainable computing practices.

A closer look at the root causes of GPU underutilization reveals several systemic challenges. First, GPUs lack virtual memory, unlike CPUs. This fundamental difference means that when a workload exceeds the available GPU memory, it triggers an out-of-memory (OOM) crash rather than utilizing overflow mechanisms like paging. This limitation significantly constrains resource sharing, as memory usage must be carefully managed to prevent crashes. Furthermore, when using GPU streams for co-running the tasks cause performance degradation due to the serialization and resource interference of the shared resources.

Observed in cases where the GPU is saturated, adding another task can lead to significant serialization and interference effects on performance. In such scenarios, exclusive execution tends to yield better results compared to collocation. NVIDIA's Multi-Instance GPU (MIG) enables rigid partitioning, ensuring an interference-free environment where each workload runs independently without resource contention. Additionally, NVIDIA's Multi-Process Service (MPS) offers an efficient solution for sharing GPU resources among multiple processes, as demonstrated in Chapter 4.

Another major factor is the layered structure of the whole ecosystem architecture. Machine learning frameworks, schedulers/resource managers, operating systems, and hardware operate in isolation, each focusing on its specific abstraction layer. For example, machine learning frameworks like PyTorch or TensorFlow are designed to abstract lower layers' complexities and provide high-level API for development and test. Meanwhile, resource managers focus on high-level orchestration, often treating GPUs and tasks as black boxes, with minimal visibility into their specific monitoring

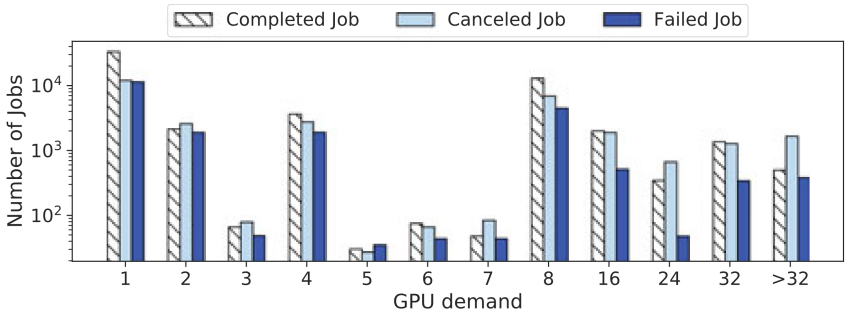


FIGURE 6.1: Distribution of requested GPU resources from studying Venus [50] (a production Deep Learning Training (DLT) cluster in SenseTime). About 52.5% of jobs use single GPU, 22.6% require 8 GPUs, and 10.3% need more than 8 GPUs. Jobs have high cancellation/failure ratio in all the cases[196]. While single-GPU jobs account for the largest proportion of all the jobs, they only consume 4.7% of total GPU service time. The major GPU service time is consumed by DLT jobs with no less than 8 GPUs (85.7%).

metrics that are collected from their hardware counters. This siloed approach creates inefficiencies, as critical information about workloads and resources is not effectively shared across layers [195].

The problem is compounded by the diversity of workloads that GPUs are tasked to handle. While much attention in research and industry is given to large-scale distributed training of massive models, the reality is that many workloads involve small to medium-sized models that run on single GPUs or server-scale setups as Figure 6.1 shows it [196]. These workloads represent a significant portion of real-world use cases, including training models for edge devices, fine-tuning pre-trained models, and experimenting with new architectures.

Current resource management systems often overlook these scenarios, leaving considerable room for optimization. Addressing these challenges requires balancing competing priorities. For example, tightly coupling solutions across abstraction layers can improve resource utilization in the short term but comes at the cost of flexibility and long-term scalability. When machine learning frameworks, schedulers, and hardware systems are deeply integrated, the resulting system risks becoming rigid and specialized, limiting its ability to adapt to new hardware or workloads. Additionally, some proposed solutions involve halting GPU kernels during execution to make collocation or scheduling decisions. While these approaches may optimize specific metrics, they introduce significant overheads, disrupt performance, and complicate system architecture. These trade-offs highlight the need for solutions that respect the independence of abstraction layers while enabling meaningful collaboration between them, and not blocking them from futuristic evolvement and innovation.

Recognizing these challenges, we propose server-level resource manager, RAD-RM, which focuses on improving GPU utilization by enabling efficient task collocation while maintaining the independence of abstraction layers and being reliable against OOM crashes. Key design principles include:

Avoiding the tightly coupled design. By steering clear of halting or modifying kernel execution, we prevent performance overheads and maintain the integrity of training workflows. This design principle leaves the system architecture layered and open to innovation. The resource manager operates independently of the framework while leveraging metadata, ensuring that each layer can evolve without imposing constraints on the others. This design philosophy not only enhances efficiency but also fosters scalability, adaptability, and creativity in future system development.

Leveraging metadata for decision-making. Instead of tightly integrating layers, our resource manager utilizes pre-computed metadata provided by the machine learning framework, enabling informed decisions without direct dependencies. For example, the parameters of the models are used as the input features for the GPU memory estimators such as GPUMemNet incorporated in RAD-RM.

Evaluation and Support for a Range of Workloads, Collocation Methods, and Policies. Our approach addresses a variety of workloads, encompassing small, medium, and large models—such as MobileNet V3 on CIFAR-100, ResNet-50 on ImageNet, and GPT-2 XL on the WikiText-2-raw-v1 dataset. These models are executed on single and dual-GPU setups, with scalability to the number of available GPUs on a server. We support various collocation methods, including streams, Multi-Process Service (MPS), and Multi-Instance GPU (MIG), acknowledging that MIG requires more administrative involvement. Additionally, our approach accommodates different collocation policies, such as exclusive (no collocation), round-robin, most available GPU memory, and least utilized GPU.

Preventing GPU overload. To aid with the scheduling decisions expected GPU memory use estimators such as GPUMemNet (Chapter 5) is used. This helps in minimizing the number of OOM crashes by providing reliable memory predictions. For managing compute interference, we compounde the GPU utilization (i.e., SMACT).

Recovery. Out-of-memory (OOM) crashes occur when a task's GPU memory demands exceed available capacity, either due to individual task requirements or memory fragmentation. Such crashes pose challenges for resource management systems that aim to collocate tasks efficiently. To address this, we propose a simple recovery method: upon detecting an OOM crash, the resource manager moves the affected task to a high-priority recovery queue. The system then assigns exclusive GPU access to these tasks, ensuring reliable execution before resuming normal operations.

We list our contributions in this chapter as follows:(1) A server-scale resource manager with collocation support of MIG, MPS, streams with multiple collocation policies (2) Integrating GPUMemNet and introducing recovery method for dealing with OOM crashes. Furthermore, a static GPU utilization thresholding for dealing with resource interference (3) Evaluating the proposed system using two traces inspired by real-world datasets [197, 196, 12]. Our evaluations show 30.13% and 13.13% improvement for total time execution and energy consumption respectively when collocation is applied. Our findings show that incorporating collocation into resource management strategies can significantly enhance GPU utilization.

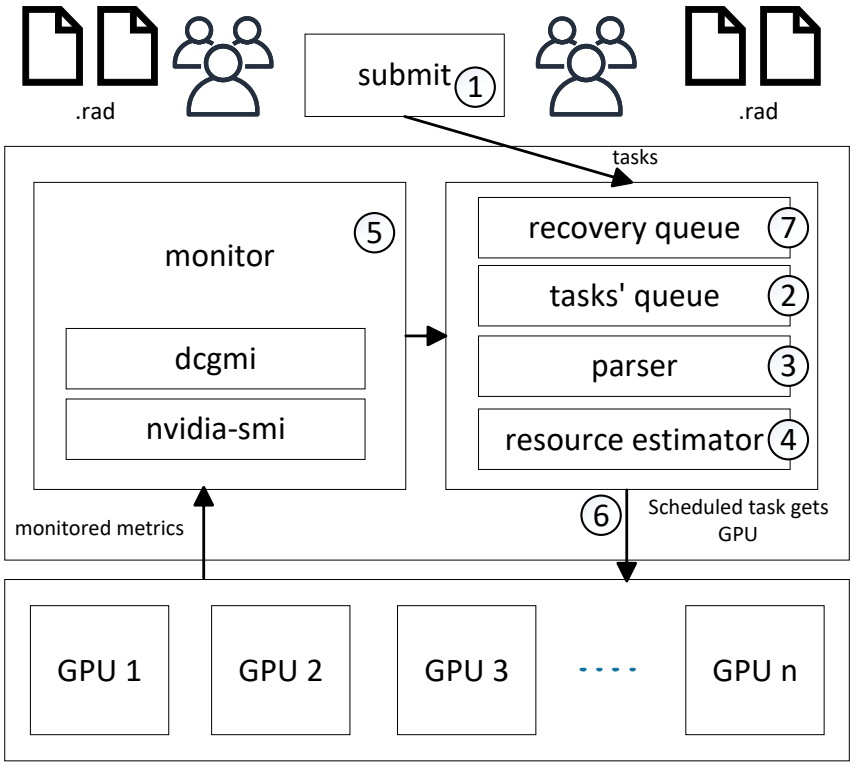


FIGURE 6.2: Overview of RAD-RM.

The remainder of this chapter is structured as follows. Section 6.2 introduces the RAD Resource Manager. In Section 6.3, we describe the setup and methodology. The results of our evaluation are discussed in Section 6.4. Finally, we conclude in Section 6.5.

6.2 RAD Resource Manager (RAD-RM)

In this section, first we explain the proposed resource management system architecture in subsection 6.2.1, then we delve into collocation policies in subsection 6.2.2. Next, we explain the recovery method for having a reliable collocation decisions in subsection 6.2.3. Finally, in Chapter 6.2.4 we explain how different collocation options are integrated into the system.

6.2.1 System Architecture

The architecture of RAD-RM is shown in Figure 6.2. The order of operations as follows:

1. User submit their tasks to the system with the help of the submit interface, the users should prepare a script, a .rad file, similar to what SLURM [42, 41] users do. The example of .rad file is shown in Listing 6.1. The submit script, gets the tasks and queues them in the tasks' queue as shown with number.

2. Tasks queue is a data structures keeping the received tasks from users.

3. Model summary parser parses the provided model summary file by the user and provides the estimator with the features (e.g., number of activations, number of parameters, number of different types of layers) for the GPU Memory Prediction.

4. The chosen GPU memory estimator (Horus method, FakeTensors, GPUMemNet, etc.) gets its input from the parser and makes an estimation for the expected GPU memory requirement of the model to be trained.

5. The monitor component monitors the relevant metrics of the hardware system including GPU memory usage with nvidia-smi monitoring interface and GPU utilization (SMACT), PCIe traffic, etc. using the dcgmi tool (Chapter 3) at configurable time intervals (e.g., every 1 seconds for 90-second time window) to provide the resource manager with the current load on all the available GPUs in the server.

6. The resource allocator (mapper) decides which GPU should be the host of a specific task that is scheduled to get resources. The current version adopts a First-In First-Out (FIFO) policy only for selecting tasks out of the tasks queue.

7. Finally, the system goes over the log files of the dispatched tasks and checks for out-of-memory keyword (i.e., OOM), since despite the high-accuracy memory estimators, this can still happen. If this is found in the logs, then the corresponding task is added to the recovery queue, which it has higher priority compared to tasks' queue. The tasks in recovery queue is mapped onto a GPU with no other tasks on to avoid further OOM scenarios for them. Afterward, other tasks can be collocated with this one, if the GPU has resources available.

Listing 6.1: .rad File Example

```
# assumption is that model.txt file is in the same directory
conda activate tf               # specifying the conda environment
python resnet.py --b 128        # command to execute
model: resnet.txt               # model summary file name
batch_size: 128                 # batch size
policy: exclusive/ collocation
```

6.2.2 Collocation Policies

To be able to make effective collocation decisions, RAD-RM must monitor the target compute and memory resources with and the jobs waiting in the tasks queue. Decisions without the knowledge on the current hardware availability will result in suboptimal collocation on GPUs, and therefore performance degradation and energy waste.

The monitor component (Figure 6.2-5) monitors the GPUs with representative metrics, like the ones highlighted in Chapters 3 and 4. The metrics we focus on are SMACT for GPU compute utilization and the GPU memory use for now. However, one can easily add additional metrics such as PCIe traffic and energy consumption if

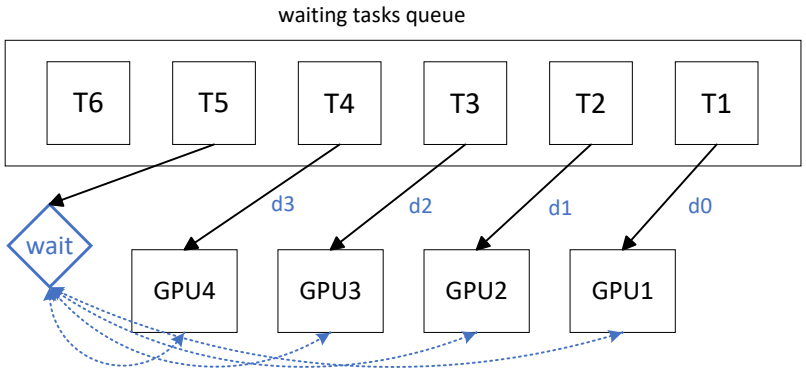


FIGURE 6.3: Exclusive policy, assigning each GPU to a single task, e.g., task T1 mapped to GPU1 at time d0, and waiting till a GPU gets free to dispatch another task for execution.

it is relevant for the performance optimization goals of the users. These metrics are monitored over a range of time (90-second time window with 1-second intervals).

The parser (Figure 6.2-3) extracts the model parameters that are relevant for the memory estimators (Figure 6.2-4) for each job waiting in the tasks queue (Figure 6.2-2) in FIFO order.

RAD-RM has several collocation policies implemented to provide the end-users with a range of options and to allow comparison across the different options. Each of these policies can be deployed with the memory estimator or without it.

Exclusive: This policy exclusively allocates the requested number of idle GPUs to the selected task. Therefore, it also represents the most conventional baseline in task scheduling, since it employs no collocation, only idle GPUs are considered. This is how the resource managers traditionally map GPUs to tasks. This policy is one of the main reasons why GPUs are suffering from underutilization for machine learning tasks since both the tasks and GPUs are considered as a black boxes. Figure 6.3 show how tasks T1 to T4 after being selected in the FIFO manner from the tasks queue in d0 to d4 moments get GPU1 to GPU4. Then, when T5 is selected by the scheduler, it needs to wait till a GPU gets idle. As the figure show, any of the GPUs getting idle sooner, will be host of T5 task.

Round-Robin (**RR**): Round-robin is a widely used resource allocation policy that ensures equitable distribution of resources among multiple entities by assigning them in a fixed, cyclic order. The primary advantage of the round-robin policy is its simplicity and fairness, as it prevents resource starvation by ensuring that each entity receives an equal share of resources in a predictable order. While making collocation decisions, RAD-RM uses round-robin as an option to determine which GPU to collocate the next task to. Going back to Figure 6.3, with this policy, T5 will be assigned to GPU1.

Most Available GPU Memory (**MAGM**): Among the target GPUs for collocating the next task in the queue, this policy first determines the GPUs with more than 5GB free memory and 20% compute resources (SMACT <= 80%) available. Then, it

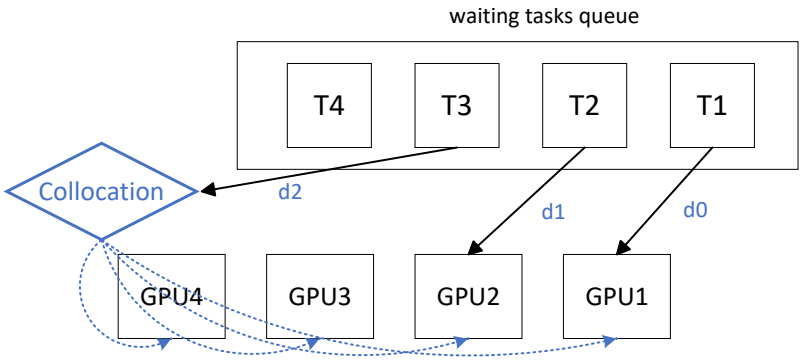


FIGURE 6.4: Filling the active GPUs first, or most utilized GPU, approach.

chooses the GPU with the highest free memory among them. Choosing the GPU with most memory available helps to minimize the probability of OOM crashes. The reason behind selecting 5GB is to aligned with the smallest memory splitting granularity that MIG instances allow (Chapter 4). The reason for 80% threshold is based on our experimentation with different values for this threshold. The reason to have the threshold in the first place is to avoid negative interference effects of overloading the GPU resources.

Least Utilized GPU (**LUG**): This policy is very similar to *MAGM*, but the GPU with the lowest SMACT value is picked instead to minimize resource interference. More specifically, among the target GPUs, this policy first determines the GPUs with more than 5GB free memory and 20% compute resources (SMACT <= 80%) available. Then, it chooses the GPU with the highest free compute resources among them.

Most Utilized GPU (**MUG**): This policy aims to consolidate as many tasks as possible onto a GPU, leaving idle GPUs unassigned as long as there are available compute and memory resources on the active GPU. Figure 6.4 shows an example of this approach. Consider when tasks T1 and T2 are executing on GPU1 and GPU2, then at d2 time point, T3 is selected for getting GPU, with the this approach, GPU1 and GPU2 are still in the list of to-check GPUs to be assigned to T3. The advantage of this approach is that when there are few tasks in the system, fewer GPUs are activated, allowing energy management policies to be applied to idle resources. This policy closely resembles solving a bin-packing problem. However, the effectiveness of this decision depends on the trade-off between the cost of the energy-scaling mechanism and the duration for which the GPUs will remain idle. For the energy-saving measures to be beneficial, the savings must outweigh the associated costs. Overall, in our earlier experiments, we found this policy to poorly perform, therefore, it is omitted from the evaluation (Section 6.4). Furthermore, even though the idle GPUs go into lower power modes, hence consuming less energy, they still consume energy due to being on. Therefore, it is better to utilize them rather than keeping them idle.

6.2.3 Recovery Method

Out of memory (OOM) crashes happens when a task needs more GPU memory than what the GPU assigned to it has. When sharing a GPU if a task is training on the GPU, and the second task joins and cannot get the amount of the GPU memory it needs, this second task will crash, while the first task keeps running. Therefore, Chapter 5 introduced GPUMemNet as a way to estimate the GPU memory needs of deep learning tasks to minimize the chances of such OOM-related crashes. RAD-RM integrates GPUMemNet and other alternative memory estimators such as the one used by Horus [61] and FakeTensors [149].

On the other hand, even if one has the perfect memory estimator and uses this knowledge in collocation decisions, OOM can still happen due to fragmentation in GPU memory allocations. For example, let's assume a scenario when the free GPU memory is fragmented in two chunks like 5GB and 4GB and a new task needs 8GB. The GPU monitors will report 9GB free GPU memory, which is what the GPU has free in total, but in two chunks. Therefore, the resource manager will map the task to that GPU, assuming it has enough free memory, but OOM crash will happen regardless for that training task.

For addressing this issue, we propose a lightweight recovery method. RAD-RM iteratively checks the error files of the tasks mapped for execution, and upon detecting OOM crashes, adds those tasks to another queue (Figure 6.2-7). This recovery queue has a higher priority compared to the main tasks queue to ensure timely rescheduling of the crashed tasks. The tasks in the recovery queue is scheduled using the exclusive policy.

6.2.4 Integration of Collocation Methods: Streams, MPS, and MIG

RAD-RM supports the three collocation methods offered by the NVIDIA GPUs: Streams, MPS, and MIG (Chapter 4).

Enabling MPS as part of the system on specified GPUs can happen when the RAD-RM launches. When the MPS is not enabled, upon collocation, NVIDIA streams will be default way of sharing the GPU.

For splitting MIG-capable GPUs, the RAD-RM does not split or unite the MIG instances automatically. It only detects them and dispatches tasks to them exclusively. However, enabling MIG even on a single A100 GPU in a server complicates the monitoring component of RAD-RM. More specifically, such a situation prevents us from monitoring non-MIG-enabled GPUs using dcgm tool as dcgm no longer shows readings for them. This poses a problem for the RAD-RM monitoring component, which relies on this tool for reading the fine-grained GPU utilization through the SMACT metric, which is then used in collocation decisions. Currently, this issue prevents us from testing scenarios, where the different GPUs in a server utilizes different collocation options (e.g., one using MIG, one MPS, and one streams) with

Software Information	
CUDA Version	12.2
PyTorch Version	2.4.1
GPU Architecture and Information	
GPU Architecture	NVIDIA Ampere
Compute Capability	8.0
Streaming multiprocessors (SMs)	108
FP32 per SM	64
Tensor Cores per SM	4
Shared Memory and L1 Cache	192KB combined configurable up to 164KB
Max 32-bit Registers per SM	64KB
L2 Cache	40MB
GPU Memory	40 GB of high-speed HBM2 memory
Collocation Capabilities	MPS and MIG
Thread and Warp Management	
Max Threads per Warp	32
Max Thread Blocks per SM	32
Max Warps per SM	64
Max Thread Block Size	1024
Max Registers per Thread	255
Host System	
CPU	AMD EPYC 7742
CPU Memory	512GB

TABLE 6.1: Software specifications of the test system (top) and hardware characteristics of NVIDIA A100 GPUs with 40GB (bottom).

RAD-RM. On the other hand, this issue is a side-effect of the immaturity of MIG-related tooling due to MIG being the newest collocation technology. We expect that this tooling will get better over time as the technology matures, and this limitation can easily be prevented.

6.3 Setup & Methodology

In this section, we outline the experimental setup used to evaluate the different collocation policies in RAD-RM organized into four parts: the hardware and software stack, the traces and models used for experimentation, and the oracles as the baselines, and the key performance metrics.

6.3.1 Hardware and Software Stack

All experiments were conducted on an NVIDIA DGX Station A100, a high-performance computing workstation designed specifically for AI tasks. The DGX Station A100 is equipped with four NVIDIA A100 GPUs, each with 40GB of high-bandwidth HBM2 memory, enabling it to handle demanding computational workloads with ease. The NVIDIA A100 GPUs are based on the Ampere architecture, featuring

advancements such as third-generation Tensor Cores and Multi-Instance GPU (MIG) technology. These innovations allow for accelerated training and inference, as well as the ability to partition each GPU into multiple smaller instances for running diverse workloads simultaneously.

As for the software framework, we use PyTorch v2.4.1 for all experiments and CUDA v 12.2.

Table 6.1 provides a comprehensive overview of the software and hardware specifications of the evaluation system.

TABLE 6.2: Models, their setup during training, and their GPU memory need in MB. The model runs shown above the line are the heavier ones compared to the ones shown below the line.

Model	Domain	Dataset	Batch Size	#GPUs	Epoch Time(m)	#Training Epochs	GPU Memory (MB)
xlnet_base	transformer	Wiki	8	2	8.949518	8	9718
BERT_base	transformer	Wiki	32	1	14.868351	1	20774
xlnet_large	transformer	Wiki	4	2	25.313793	3	14546
mobilenet_v2	CNN	ImageNet	128	1	34.910391	1	12578
resnet50	CNN	ImageNet	128	1	35.007046	1	15120
efficientnet_b0	CNN	ImageNet	128	1	35.211261	1	13832
efficientnet_b0	CNN	ImageNet	64	1	35.412238	1	7844
mobilenet_v2	CNN	ImageNet	64	1	35.428402	1	7224
resnet50	CNN	ImageNet	64	1	35.502900	1	8536
mobilenet_v2	CNN	ImageNet	32	1	36.088235	1	4544
efficientnet_b0	CNN	ImageNet	32	1	36.210931	1	4958
resnet50	CNN	ImageNet	32	1	36.315977	1	5262
vgg16	CNN	ImageNet	128	1	42.418796	1	24408
vgg16	CNN	ImageNet	64	1	44.381296	1	13642
Xception	CNN	ImageNet	128	1	44.440120	1	22978
inception	CNN	ImageNet	128	1	44.853645	1	19018
BERT_large	transformer	Wiki	8	1	44.932988	1	13568
Xception	CNN	ImageNet	64	1	45.779347	1	11520
inception	CNN	ImageNet	64	1	46.290656	1	10560
Xception	CNN	ImageNet	32	1	46.863864	1	7202
vgg16	CNN	ImageNet	32	1	48.448318	1	8222
inception	CNN	ImageNet	32	1	50.103753	1	6346
gpt2_large	transformer	Wiki	8	2	64.962650	1	27902
efficientnet_b0	CNN	cifar100	32	1	0.766600	20, 50	1858
efficientnet_b0	CNN	cifar100	64	1	0.480000	20, 50	1912
efficientnet_b0	CNN	cifar100	128	1	0.273400	20, 50	2054
resnet18	CNN	cifar100	32	1	0.328400	20, 50	1960
resnet18	CNN	cifar100	64	1	0.220000	20, 50	1966
resnet18	CNN	cifar100	128	1	0.163400	20, 50	2006
resnet34	CNN	cifar100	32	1	0.486817	20, 50	2152
resnet34	CNN	cifar100	64	1	0.302731	20, 50	2168
resnet34	CNN	cifar100	128	1	0.200908	20, 50	2190
S mobilenetv3	CNN	cifar100	32	1	0.536289	20, 50	1784
S mobilenetv3	CNN	cifar100	64	1	0.322098	20, 50	1790
S mobilenetv3	CNN	cifar100	128	1	0.217667	20, 50	1824

6.3.2 Trace & Models

To mimic real-world deep learning training job/task traces, we use the trace shared by the authors of [92], the *Microsoft Philly Trace*. We create two traces for our experiments, one with 60 tasks and one with 90 tasks, where the submission times of the tasks match the submission times of a subset of the tasks from the Microsoft trace from a time window. Since the Microsoft trace is from a cluster of machines, while

our experiments run on a single server, we use a trimmed version of the whole trace from the chosen time window. Otherwise, the total number of requests submitted to the single server would be unrealistically high.

Since the trace does not disclose the model types, we pick the model types and configurations ourselves based on the real-world task sizes and time distribution from [196]. Table 6.2 lists the models and each different configuration we run them with to mimic a diverse range of options in terms of GPU utilization, GPU memory requirements, and execution times. The 60-task trace is only composed of medium and large models (83% medium and 17% large models) from the list in Table 6.2 to serve as a stress-test for collocation, whereas 90-task trace is mostly lighter models (65% light, 27% medium, 8% large models) that would benefit more easily from collocation. However, for the execution time of the tasks, we followed the real-world tasks length analyzed in [196].

6.3.3 Oracle Baselines

To establish baselines for the overall performance of the collocation mechanisms, we deploy the same traces while providing the resource manager with the exact GPU memory requirements of the models. We refer to this scenario as the *oracle* case, as the resource manager is equipped with complete knowledge to prevent OOM crashes. However, even with accurate GPU memory requirements, the task-to-GPU mapping process requires a policy that leverages the available knowledge about the tasks and the hardware state. We considered the following policies for these *oracle* runs. All the oracle policies operates on the two common principles: (1) there should be at least 2GB extra for the available GPU memory in addition to the memory need of the deep learning task to be scheduled to prevent OOM crashes due to GPU memory fragmentation, and (2) the target GPU's SMACT value should be less than 80% to prevent negative interference and GPU compute overload.

Oracle First Fit (**OFF**): This policy first identifies the GPUs that match the required memory and SMACT conditions for the deep learning task to be scheduled. Then, picks the first GPU in the list to map the task onto that GPU.

Oracle Best Fit (**OBF**): This policy first identifies the GPUs that match the required memory and SMACT conditions for the deep learning task to be scheduled. Then, picks the GPU with the least available memory in that list to first pack as much as possible to one GPU before moving onto others. Thus, it aligns with the *Most Utilized GPU (MUG)* policy described in Section 6.2.2.

Oracle Worst Fit (**OWF**): This policy first identifies the GPUs that match the required memory and SMACT conditions for the deep learning task to be scheduled. Then, picks the GPU with the most available memory in that list to spread out the load. Thus, it aligns with the *Most Available GPU Memory (MAGM)* policy described in Section 6.2.2.

Oracle Least Utilized GPU (**OLUG**): This policy first identifies the GPUs that match the required memory and SMACT conditions for the deep learning task to be

scheduled. Then, picks the GPU with the most available compute resources, lowest SMACT value, to prevent overload and interference. Thus, it aligns with the *Least Utilized GPU (LUG)* policy described in Section 6.2.2.

6.3.4 Evaluation Metrics

Trace Total Time

This metric represents the elapsed time from the moment the first task is received and queued until all tasks have been completed and retired from the server.

Average Waiting Time

This metric represents the average time a task spends waiting in the queue before it begins execution on the server. It is calculated as the total waiting time of all tasks divided by the number of tasks in the trace.

Average Execution Time

This metric represents the average time a task spends in execution on the server. It is calculated as the total execution time of all tasks divided by the number of tasks in the trace.

Average Job Completion Time (JCT)

This metric refers to the average time taken for a job to be fully completed, from the moment it is submitted to the system until its execution finishes. It is calculated by dividing the total completion time of all jobs by the number of jobs in the trace.

GPU Memory Usage

This metric represents the amount of GPU memory allocated during task execution, measured by `nvidia_smi`.

GPU Utilization (SMACT)

This metric quantifies the activity level of a GPU's Streaming Multiprocessors (SMs), measured by `dcgmi`. SMACT represents the fraction of time during which at least one warp (a group of threads) is active on an SM, averaged across all SMs. A higher SMACT value indicates more intensive utilization of the GPU's computational resources, while lower values may suggest underutilization or idle periods. Monitoring SMACT is essential for assessing the efficiency of GPU workloads and identifying potential performance bottlenecks.

GPU Power (W)

This metric measures the power consumption of a GPU in watts (W) during its operation, measured by `dcgmi`.

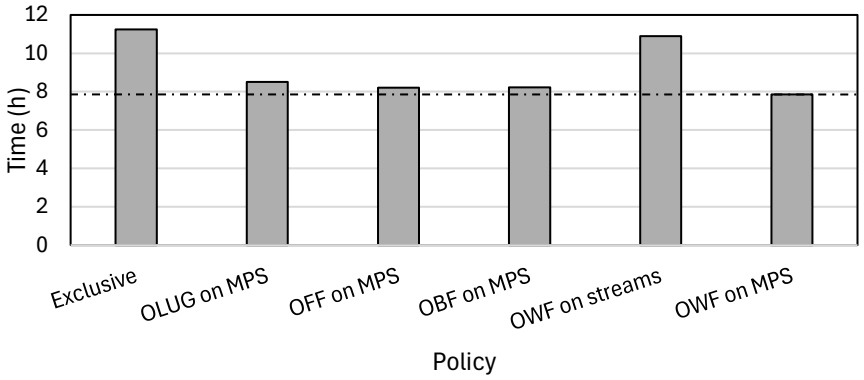


FIGURE 6.5: Oracle Total Trace Time Comparison for 90-task Trace (conditions are SMACT remain under 80% and 2GB safety margin for GPU memory).

GPU Energy Consumption (MJ)

we assess GPU energy consumption in megajoules (MJ) by utilizing the DCGM tool. DCGM provides a metric, which reports the total energy consumed by the GPU since boot, measured in millijoules (mJ). By capturing this metric at the start and end of our measurement period, we calculate the total energy consumption during that interval.

Number of Out-of-Memory Crashes

We quantify the frequency of out-of-memory (OOM) crashes, which occur when a GPU exhausts its available memory, leading to the task's termination. To measure this, we monitor the dispatched tasks' error logs on each decision making iteration. By analyzing these logs, we can determine the number of OOM crashes that have occurred during our testing period.

6.4 Results

In this section, we evaluate RAD-RM under a variety of scenarios and policies. First Section 6.4.1 evaluates the proposed collocation policies in oracles cases (assuming we know the exact GPU memory need - Section 6.3.3) Then, Section 6.4.2 evaluates the policies proposed in Section 6.2.2 without using any knowledge of or estimators for the GPU memory requirements of the tasks, followed by Section 6.4.3 that evaluates the same policies while using the memory estimators. All of the aforementioned analyses are for the 90-task trace that has a higher percentage of deep learning training jobs that would benefit from collocation. To observe the impact of heavier jobs on collocation, Section 6.4.4 repeats the experiments from Section 6.4.3 with the 60-task trace. Finally, Section 6.4.5 shows GPU utilization and power consumption over time in exclusive mode compared to the best performing collocation scenario to show the gains, and Section 6.4.6 delves into the saved GPU energy consumption by adopting collocation.

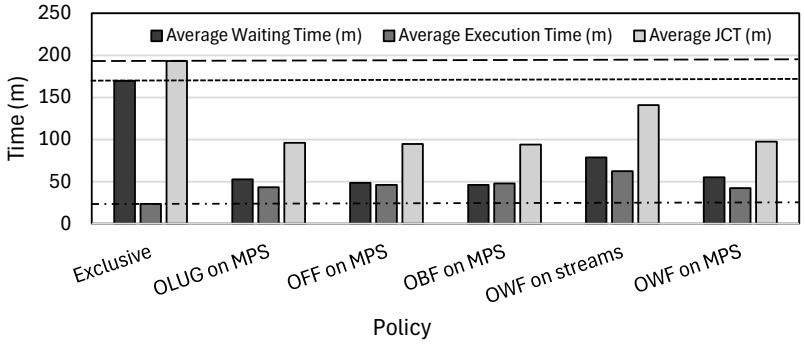


FIGURE 6.6: Oracle Average Waiting Time, Average Execution Time, Average JCT (conditions are SMACT remain under 80% and 2GB safety margin for GPU memory).

6.4.1 Oracle cases

Figure 6.5 show the comparison of total end-to-end execution time of the 90-task trace. The results show *worst-fit (OWF)*, picking the GPUs with most GPU memory available, with MPS-based collocation outperforms the rest of the solutions. On the other hand, collocation with NVIDIA streams provides only marginal benefits over *exclusive* execution. This results align with the conclusions of Chapter 4 when it comes to the effectiveness of these two collocation methods. For the rest of this section, we will therefore present the results for MPS-based collocation, unless stated otherwise.

Figure 6.6 shows how GPU resource interference causes slowdowns in cases when the workloads are tightly packed only based on their GPU memory requirement to be able to serve as many as possible training tasks. This is illustrated by the higher average execution time exhibited by the *best-fit (OBF)* policy. In contrast, we observe **30.13%** improvement when MPS-based collocation is coupled with, *worst-fit (OWF)*, aligning with *Most Available GPU Memory (MAGM)* policy. Furthermore, it is noteworthy to mention that while collocation on streams does not benefit in terms of the total time execution of the whole trace, it improves average waiting time that contributes to the average JCT.

Policy	**#OOM Crashes**
RR (no condition)	8
MAGM (no condition)	5
MAGM (SMACT $\leq$ 80%)	4
MAGM (SMACT $\leq$ 80% and GMem $\geq$ 2GB)	2
MAGM (SMACT $\leq$ 80% and GMem $\geq$ 5GB)	2
LGU (SMACT $\leq$ 80% and GMem $\geq$ 5GB)	2
MAGM (SMACT $\leq$ 75% and GMem $\geq$ 5GB)	1
MAGM (SMACT $\leq$ 85% and GMem $\geq$ 5GB)	2

TABLE 6.3: Total number of out-of-memory (OOM) errors when the collocation policies rely only on the basic recovery method described in Section 6.2.3.

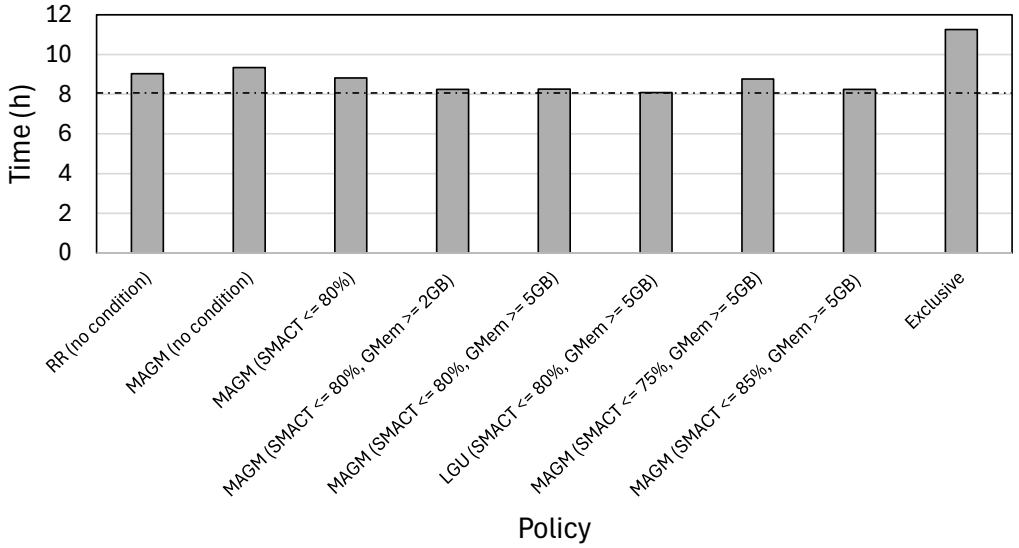


FIGURE 6.7: Total trace time comparison for 90-task Trace with different collocation policies when GPU memory requirements are neither known nor predicted. All collocation runs use MPS.

6.4.2 Relying Only on the Recovery Method for OOM

These series of experiments assume that we do not know anything about the GPU memory need of different tasks. As a result, we collocate till an OOM(s) crash happens and the recovery mechanism (from Section 6.2.3) takes care of the situation. As Table 6.3 shows one can add collocation conditions to avoid OOM crashes by filtering out GPUs with low available GPU memory. For example, *Most Available GPU Memory (MAGM)* policy with SMACT remaining less than 75% and GPU memory more than 5GB, limits the number of OOM crashes to only one crash.

While these conditions may take away the potential benefits from collocation, applying them are necessary for better performance. When we do not apply such conditions, due to the increased waiting time when OOM happens, we observe that *Round-robin (RR)* and *MAGM* without any condition offer the worst performance for trace end-to-end execution time in Figure 6.7. The figure shows that *Least Utilized GPU (LUG)* policy with SMACT remaining under 80% and having at least 5GB GPU memory offers the best performance with 28% improvement in end-to-end execution time.

On the other hand, looking at the different ways of limiting SMACT with *MAGM*, while limiting SMACT to be less than 75 eliminates one more OOM compared to limiting SMACT to be less than 80 or 85, it takes collocation potential away as Figure 6.8 shows in the increase in job waiting time.

Nevertheless, these results highlight the importance and benefits of deploying a lightweight recovery method against OOMs to ensure robust execution of deep learning tasks.

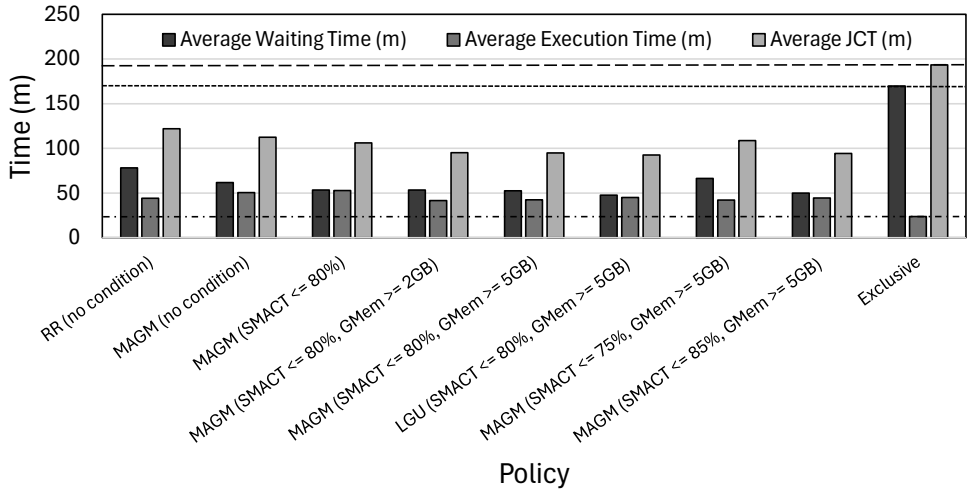


FIGURE 6.8: Average Waiting Time, Average Execution Time, Average JCT with with different collocation policies when GPU memory requirements are neither known nor predicted, and the basic recovery method is used against OOM. All collocation runs use MPS.

Policy	**#OOM Crashes**
MAGM (GMem $\geq$ Horus)	1
MAGM (GMem $\geq$ faketensor)	0
MAGM (GMem $\geq$ GPUMemNet)	1
MAGM (SMACT $\leq$ 80 and GMem $\geq$ Horus)	0
MAGM (SMACT $\leq$ 80 and GMem $\geq$ faketensor)	0
MAGM (SMACT $\leq$ 80 and GMem $\geq$ GPUMemNet)	0

TABLE 6.4: Total number of out-of-memory (OOM) errors when we integrate the GPU memory estimators (Chapter 5) into RAD-RM while applying the *Most Available GPU Memory (MAGM)* collocation policy. The memory estimators manage to minimize, mostly eliminate, the OOM errors.

6.4.3 GPU Memory Estimators into Action

Next, we evaluate the impact of the GPU memory estimators for deep learning training tasks. We only evaluate the *Most Available GPU Memory (MAGM)* collocation policy here because of its overall good performance in earlier results.

Table 6.4 demonstrates that applying collocation with GPU memory estimation (almost) eliminates the OOM crashes.

Figures 6.9 and 6.10 show that *MAGM* with GPUMemNet estimator offers the best performance with around 25% improvement compared to *exclusive* execution. Overall, the results highlights the benefits of memory predictors, even though the clear winner in terms of performance may change based on the workload traces.

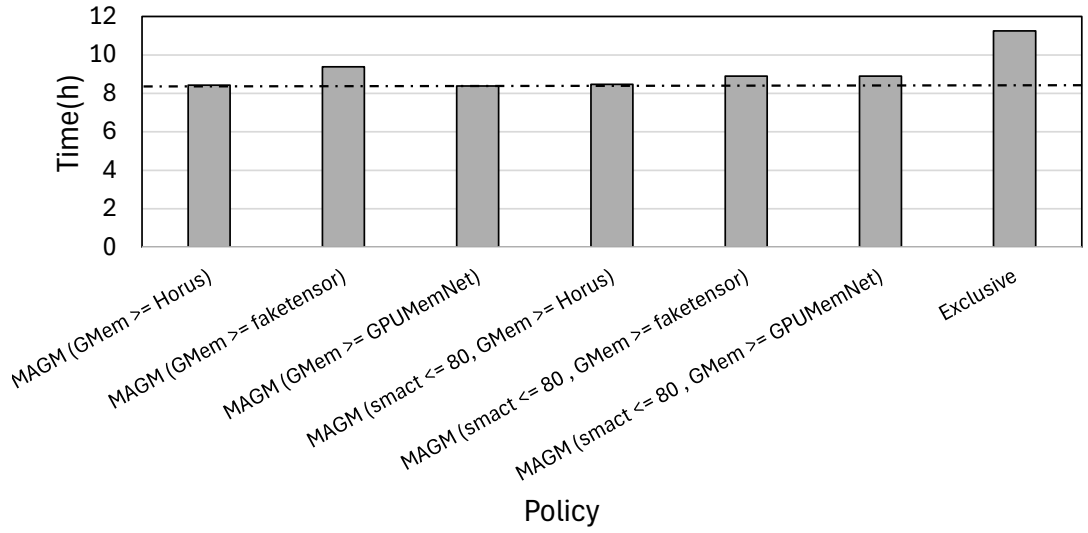


FIGURE 6.9: Total trace time comparison for 90-task trace with *Most Available GPU Memory (MAGM)* collocation policy using different GPU memory estimators. All collocation runs use MPS.

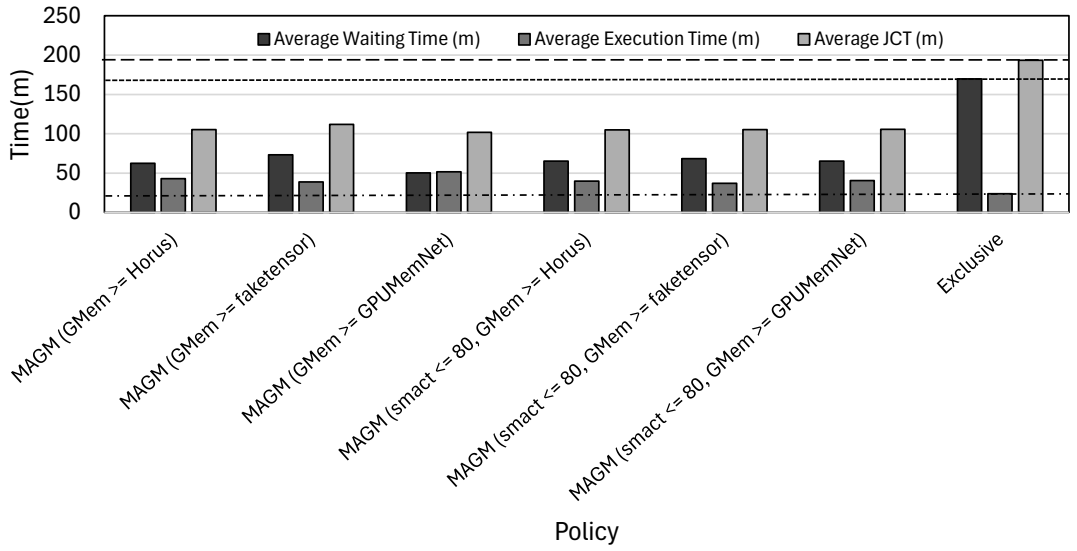


FIGURE 6.10: Average Waiting Time, Average Execution Time, Average JCT with GPU memory estimators integrated in RAD-RM using *Most Available GPU Memory (MAGM)* collocation policy and MPS.

Experiment	**#OOM Crashes**
Exclusive (exclusive)	0
Round Robin on streams + Recovery	9
Round Robin on MPS + Recovery	6
MAGM (2GB, 80%) + Recovery	4
LGU (2GB, 80%) + Recovery	4
MAGM + Horus (80%)	2
MAGM + FakeTensor (80%)	3
MAGM + GPUMemNet (80%)	1

TABLE 6.5: Total number of out-of-memory (OOM) errors for the heavier 60-task trace for different collocation policies when the memory estimators are integrated within RAD-RM.

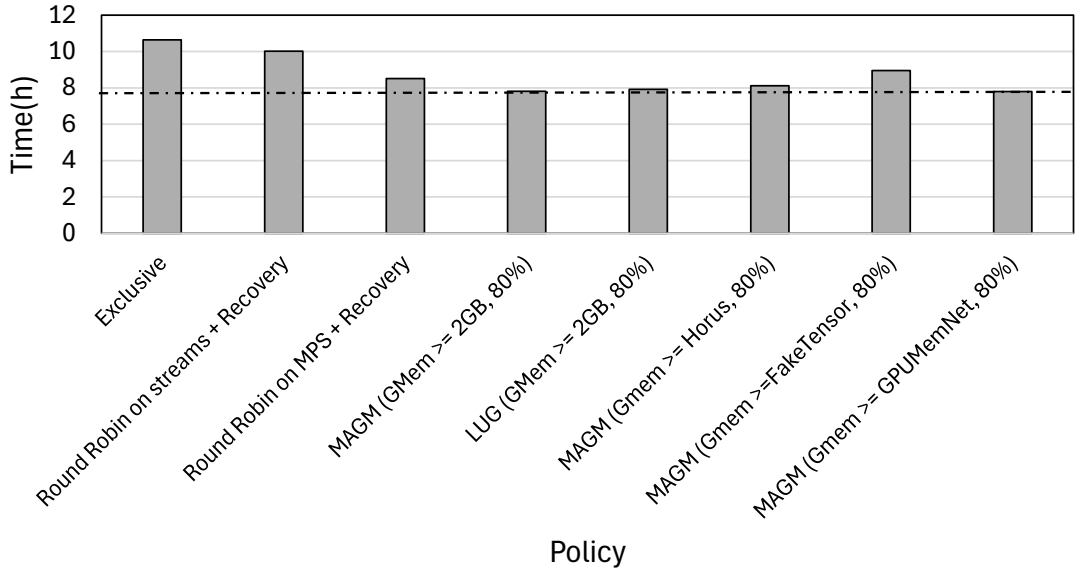


FIGURE 6.11: Total trace time for 60-task trace with different collocation policies and GPU memory estimators.

6.4.4 Trace of 60-tasks

The trace with the 60-tasks (Section 6.3.2) is overall heavier than the 90-task trace, since it involves a higher percentage of the heavier model training tasks listed in Table 6.2. Table 6.5 shows how effective GPUMemNet can be in avoiding OOM crashes for this trace. Furthermore, Figures 6.11 and 6.12 show the benefits of collocation even for this heavier workload trace.

6.4.5 GPU Metrics Over Time

In this section, we look into the impact of collocation on GPUs memory usage, utilization, power over time in comparison to the exclusive scheduling of tasks.

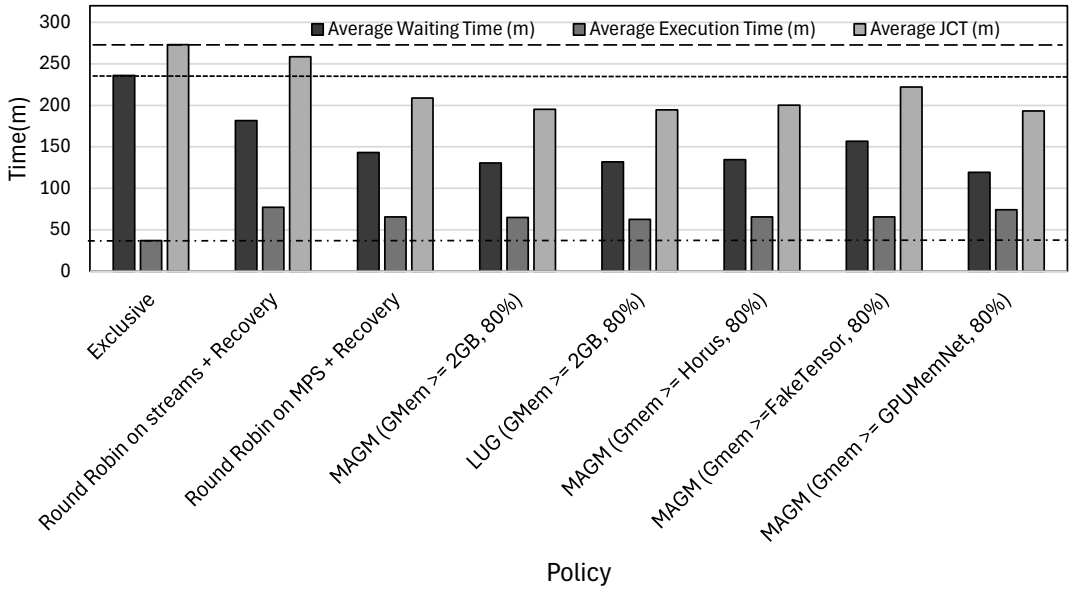


FIGURE 6.12: Average Waiting Time, Average Execution Time, Average JCT for 60-task trace with different collocation policies and GPU memory estimators.

GPU Memory Usage

Figure 6.13 shows how GPU memory gets allocated as we run the 60-task trace managed with *exclusive* and *most available GPU memory (MAGM)* with GPUMemNet estimator and MPS enabled. It shows how our proposed collocation mechanism, improves performance by shortening the end-to-end execution time of the trace and increases all GPUs memory usage, and utilization, over the execution of those tasks by collocating.

GPU Utilization

Figure 6.14 shows the GPUs' utilization over time when 60-task trace is run based on either *exclusive* and our collocation policy of *MAGM + GPUMemNet estimator while MPS enabled*. It shows how the collocation policy improves performance by shortening the end-to-end execution time of the trace and increases all GPUs utilization, hence tackling the under-utilization problem, over the execution of the tasks.

GPU Power

Figure 6.15 shows the power consumption of the different GPUs over time when 60-task trace is run based on either *exclusive* and our collocation policy of *MAGM + GPUMemNet estimator while MPS enabled*. The figure follows the same trends with GPU utilization. Section 6.4.6 shows how much GPU energy consumption gets saved because of the overall shorter execution time despite the higher power consumption per unit of time.

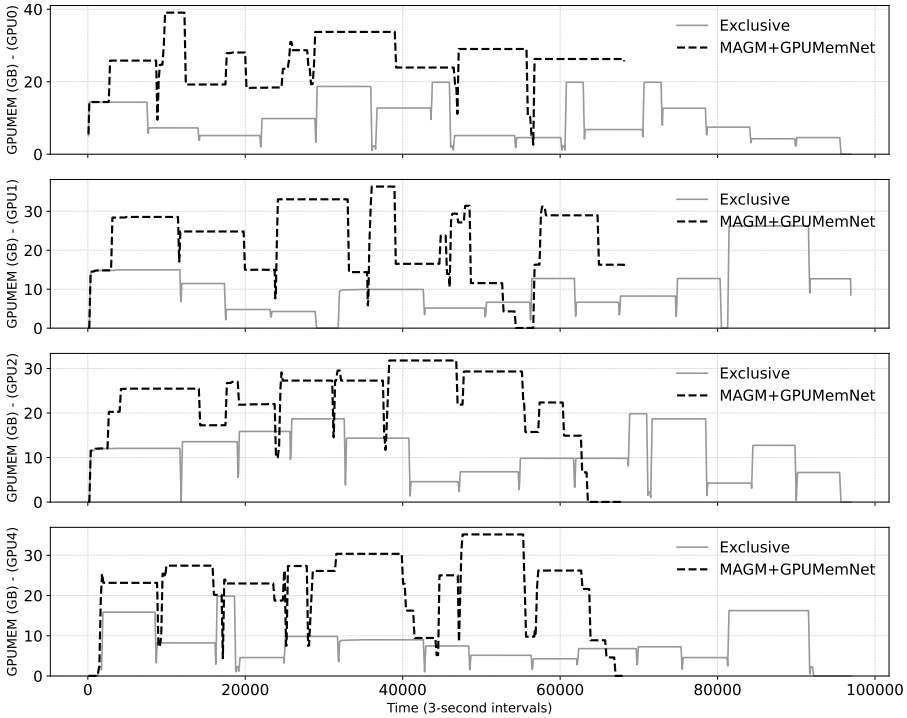


FIGURE 6.13: GPU memory allocation over time on all four GPUs on the NVIDIA DGX Station with *Exclusive* and *MAGM* policies on the 60-task trace. *MAGM* uses MPS-based collocation and GPUMemNet memory prediction.

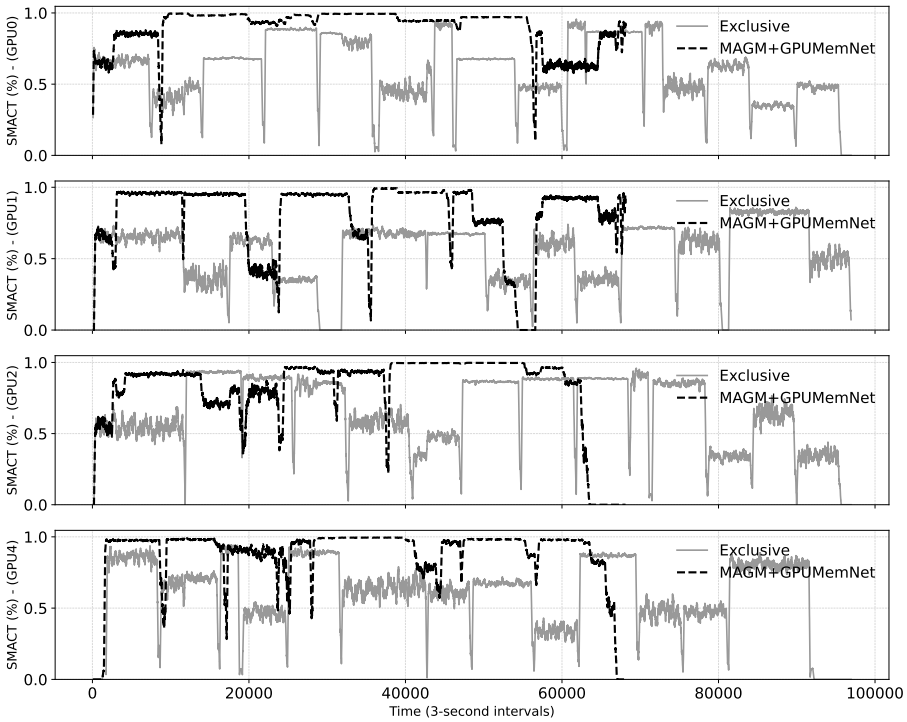


FIGURE 6.14: GPU utilization, SMACT, over time on all four GPUs on the NVIDIA DGX Station with *Exclusive* and *MAGM* policies on the 60-task trace. *MAGM* uses MPS-based collocation and GPUMemNet memory prediction.

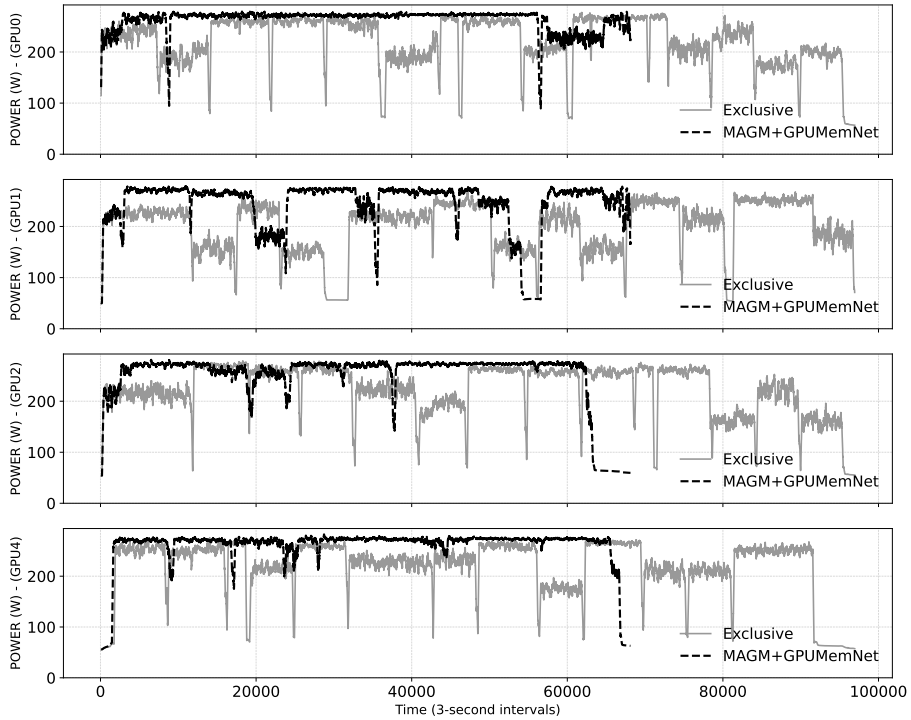


FIGURE 6.15: GPU power over time over time on all four GPUs on the NVIDIA DGX Station with *Exclusive* and *MAGM* policies on the 60-task trace. *MAGM* uses MPS-based collocation and GPUMemNet memory prediction.

6.4.6 GPU Energy Consumption

Table 6.6 reports the accumulated energy consumption of all 4 GPUs for executing the whole of 60-task trace under different policies. We report energy consumption in **megajoules (MJ)**, where 1 MJ $= 10^6$ joules. Training in *exclusive* mode consumes 33.2 MJ, whereas **MAGM + GPUMemNet on MPS** reduces this to 28.8 MJ, achieving a reduction of 4.4 MJ (**13.25%** improvement). These findings demonstrate how GPU memory estimation and efficient collocation-aware resource management can reduce energy costs in large-scale deep learning training.

Policy	**Energy Consumption (MJ)**
Exclusive	33.197904654
Round Robin on Streams	34.745196574
Round Robin on MPS	29.601240054
MAGM on MPS	28.781487533
MAGM + Horus on MPS	29.041723267
MAGM + FakeTensor on MPS	30.306237499
MAGM + GPUMemNet on MPS	28.502464491

TABLE 6.6: Energy Consumption under different policies.

6.5 Conclusion

In this chapter, we designed, implemented, and evaluated an efficient collocation-aware resource manager. We observed that GPU memory estimators are helpful for avoiding the out-of-memory (OOM) crashes and improving GPU utilization. Our evaluation demonstrated that collocating tasks effectively can enhance overall system efficiency while maintaining system reliability.

Key takeaways include:

- Collocation with proper policies significantly improves GPU utilization. Our results show up to a 30.13% improvement in overall job completion time (JCT) when using the most available GPU memory (MAGM) policy with NVIDIA MPS.

- GPU memory estimation is crucial for reliable collocation. Without estimation, OOM crashes are frequent and require a recovery mechanism. Incorporating estimators like GPUMemNet reduced crashes and further optimized resource allocation.

- On the other hand, a recovery mechanism is a must even when GPU memory estimation is adopted with the collocation policies due to the imperfect estimators and the fragmentation in GPU memory management.

- Furthermore, resource interference must be managed carefully. Policies that prioritize minimizing GPU utilization interference (e.g., least utilized GPU (LUG) with SMACT threshold) provide a balance between performance and stability.

- MPS enhances collocation performance. MPS significantly improves collocation efficiency compared to using CUDA streams alone, reducing execution overhead and increasing parallel task execution.

- Collocation leads to energy savings. Our experiments showed that enabling collocation-aware scheduling reduced total energy consumption by 13.25%, demonstrating the potential for more sustainable AI workloads.

Chapter 7

Future Directions and Conclusion

Over the past decade, data science has experienced unprecedented expansion, largely fueled by three key factors: the ever-growing volume of available datasets, continuous improvements in computational power, and ongoing advancements in learning and analysis algorithms. Deep learning has played a crucial role in this evolution, significantly increasing the demand for computational support during model training. To meet this rising demand, enterprises typically operate shared GPU clusters across multiple production teams. However, despite the abundance of resources, such clusters often suffer from underutilized GPUs.

In this concluding chapter, we discuss our visions, and offer insights into future directions, including scaling our design to cluster-level architectures, and finally we conclude with a short summary of this thesis.

7.1 GPU Utilization Estimation

In this work, we demonstrated that estimating GPU memory requirements for deep learning training tasks, combined with a collocation-aware resource manager, significantly enhances performance and energy efficiency. We further posit that complementing GPU memory estimation with GPU utilization estimation (e.g., SMACT) can minimize interference by enabling more informed decisions based on both memory and utilization metrics.

Nevertheless, a key challenge lies in the fact that SMACT is a relative measure. For example, a SMACT value of 80% on an NVIDIA A100 may represent only 60% utilization on an H200, complicating direct comparisons. Developing a machine learning model to estimate GPU utilization across diverse GPU models also poses significant hurdles, primarily due to the need for comprehensive datasets spanning various architectures. A practical approach might involve building a dataset for one GPU architecture and then extrapolating to other GPU architectures based on resource count differences.

Moving forward, we believe that adopting a more representative metric—one that accounts for architectural differences among GPUs—can lead to better standardized measures of hardware unit utilization. Such a metric would offer greater accuracy in

workload distribution and more effective resource allocation across different hardware platforms.

7.2 Looser Recovery Methods

To prevent collocation decisions from introducing additional uncertainty, we proposed a simple yet effective recovery method for tasks that crash or fail to run under their assigned resources. Rather than waiting for a GPU to become fully idle before reassigning the crashed task, a more efficient approach iteratively might seek a GPU offering, for instance, at least 5GB more available memory than the previous allocation or the estimated requirement (when an estimator is used).

This method can operate in a loop, gradually increasing the memory threshold to find a suitable GPU. If no GPU meets the updated threshold, the process continues until an idle GPU is finally discovered. By dynamically reallocating resources in this manner, we envision the system recovering more efficiently and reducing potential disruptions to other ongoing tasks.

7.3 Fairness and Checkpointing

Ensuring fairness in shared computing systems is inherently challenging, particularly when users must specify their requested runtime. Although this practice can improve scheduling efficiency, users often overestimate or underestimate their actual needs—especially in the early stages of development or during production. In systems where users do not specify runtime or operate without strict time limits, additional problems arise:

- **Resource Monopolization**: Long-running tasks may monopolize GPU resources, causing starvation for shorter jobs.
- **Inefficient Scheduling**: Inaccurate runtime estimates reduce scheduling efficiency, causing waste and necessitating job resubmission. For example, a machine learning practitioner training a model for two days might fail to reach a desired accuracy if the allocated time expires without checkpointing, effectively wasting the GPUs' resources.

To tackle these challenges, we envision a scheduling approach that integrates *configurable fixed time slots* with a robust *checkpointing mechanism*, ensuring ease of use, performance, and energy efficiency. Our two central design choices include:

1. **Checkpointing as Standard Practice**: Encouraging developers to adopt checkpointing consistently to preserve task progress and enable seamless resumption.
2. **Communication Between Frameworks and the Resource Manager**: Allowing the resource manager to coordinate checkpoint signals with machine learning frameworks at critical points.

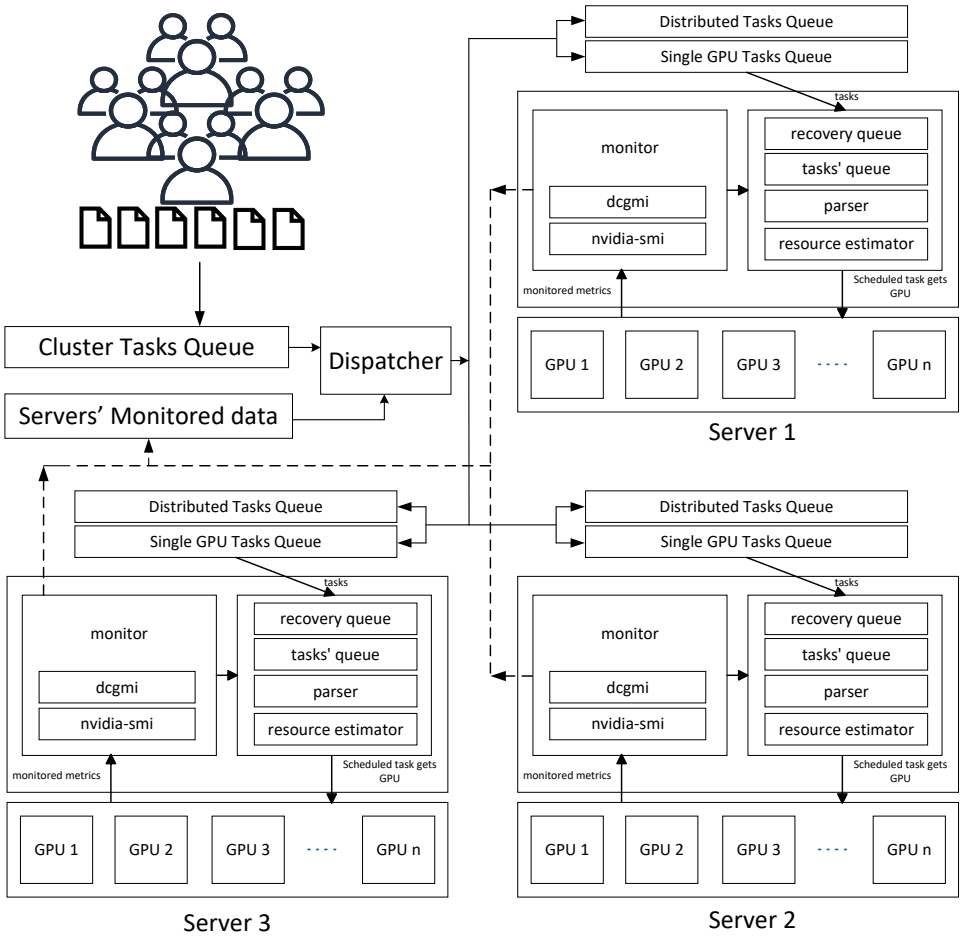


FIGURE 7.1: Vision of the extended version of server-scale resource manager to cluster-level

For instance, a 12-hour time slot could be configured for scheduled tasks. If a task finishes earlier, the monitoring system detects this and quickly reallocates the freed resources to another job. If a task is approaching the end of its time slot (e.g., 10 minutes remaining), the resource manager signals the task to begin checkpointing. Once the time slot concludes, the next job starts immediately, thereby ensuring fairness.

7.4 Extending the Design to Cluster-Scale

Building on our server-scale approach, we propose extending the design to a *cluster-scale environment* in a hierarchical manner. At each server node, local queues can manage incoming tasks by using both GPU memory and utilization estimations to make informed collocation decisions using a policy such as Most available GPU Memory. Additionally, a higher-level queue can handle distributed training workloads across multiple nodes, ensuring that large-scale, multi-GPU jobs are optimally integrated with single-node tasks. This hierarchical structure promotes scalability and balances the needs of diverse workloads more effectively.

7.5 Thesis Summary

In this thesis, we established an experimental environment and selected monitoring tools and metrics to evaluate different collocation configurations on NVIDIA GPUs. We investigated native capabilities—GPU streams, Multi-Process Service (MPS), and Multi-Instance GPU (MIG)—for running multiple deep learning training tasks concurrently. Subsequently, we introduced a machine learning–driven methodology to estimate GPU memory consumption for deep learning training models. These estimations allow the server resource manager to allocate tasks more effectively, improving overall performance and reducing energy consumption while mitigating the overhead of collocation.

In the future, through combining the proposals of this thesis with a more representative utilization metric, looser recovery strategies, fair scheduling with checkpointing, and plans for hierarchical cluster-scale deployment, we envision a more efficient, reliable, and user-friendly deep learning infrastructure for every user.

References

[1] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. URL: https://doi.org/10.1145/3065386.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS’12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[4] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[5] Avner May et al. *Kernel Approximation Methods for Speech Recognition*. 2017. arXiv: 1701.03577 [stat.ML].

[6] Hongyu Zhu et al. “TBD: Benchmarking and Analyzing Deep Neural Network Training”. In: *CoRR* abs/1803.06905 (2018). arXiv: 1803.06905. URL: http://arxiv.org/abs/1803.06905.

[7] Joel Hestness et al. “Deep learning scaling is predictable, empirically”. In: *arXiv preprint arXiv:1712.00409* (2017).

[8] Vishakh Hegde and Sheema Usmani. “Parallel and distributed deep learning”. In: *May* 31 (2016), pp. 1–8.

[9] OpenAI. *AI and Compute*. https://openai.com/blog/ai-and-compute/. Accessed: 2021-12-13.

[10] NVIDIA Corporation. *NVIDIA’s Next Generation CUDA Compute Architecture: Fermi*. Accessed: 2025-01-27. 2009. URL: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[11] NVIDIA Corporation. *NVIDIA H200 Tensor Core GPU Datasheet*. Accessed: 2025-01-27. 2024. URL: https://resources.nvidia.com/en-us-data-center-overview-mc/en-us-data-center-overview/hpc-datasheet-sc23-h200.

[12] Myeongjae Jeon et al. “Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads”. In: *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC ’19. Renton, WA, USA: USENIX Association, 2019, pp. 947–960. ISBN: 9781939133038.

[13] Cem Dilmegani. *Wu Dao 2.0: China's Improved Version of GPT-3*. https://research.aimultiple.com/wu-dao/. Accessed: 2022-12-12.

[14] NVIDIA Corporation. *Multi-Process Service*. Accessed: 2025-01-27. 2025. URL: https://docs.nvidia.com/deploy/mps/index.html.

[15] Jingoo Han et al. "Marble: A multi-gpu aware job scheduler for deep learning on hpc systems". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 272–281.

[16] Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, and Pinar Tözün. "Profiling and Monitoring Deep Learning Training Tasks". In: *Proceedings of the 3rd Workshop on Machine Learning and Systems, EuroMLSys 2023, Rome, Italy, 8 May 2023*. Ed. by Eiko Yoneki and Luigi Nardi. ACM, 2023, pp. 18–25. DOI: 10.1145/3578356.3592589. URL: https://doi.org/10.1145/3578356.3592589.

[17] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. *An Analysis of Collocation on GPUs for Deep Learning Training*. 2024.

[18] Bishwo Adhikari and Heikki Huttunen. "Iterative bounding box annotation for object detection". In: *2020 25th International Conference on Pattern Recognition (ICPR)*. IEEE. 2021, pp. 4040–4046.

[19] Jindong Wang et al. "Generalizing to unseen domains: A survey on domain generalization". In: *IEEE transactions on knowledge and data engineering* 35.8 (2022), pp. 8052–8072.

[20] Yuxin Wang et al. "Benchmarking the performance and energy efficiency of AI accelerators for AI training". In: *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE. 2020, pp. 744–751.

[21] Ian Goodfellow et al. "Generative adversarial networks". In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[22] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and policy considerations for modern deep learning research". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 09. 2020, pp. 13693–13696.

[23] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: *Journal of machine learning research* 18.153 (2018), pp. 1–43.

[24] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. 2019, pp. 8024–8035.

[25] Martin Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *OSDI 16*. 2016, pp. 265–283.

[26] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. Version 0.3.13. 2018. URL: http://github.com/jax-ml/jax.

[27] Tianqi Chen et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (2015).

[28] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *NIPS*. 2019, pp. 8026–8037.

[29] Francois Chollet et al. *Keras*. https://github.com/keras-team/keras. 2015.

[30] Rami Al-Rfou et al. "Theano: A Python framework for fast computation of mathematical expressions". In: *arXiv e-prints* (2016), arXiv–1605.

[31] Frank Seide and Amit Agarwal. "CNTK: Microsoft's open-source deep-learning toolkit". In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 2135–2135.

[32] Richard Vuduc and Jee Whan Choi. "A Brief History and Introduction to GPGPU". In: *Modern Accelerator Technologies for Geographic Information Science*. Springer, 2013, pp. 9–32. DOI: 10.1007/978-1-4614-8745-6_2.

[33] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. 2007. URL: https://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[34] NVIDIA Corporation. *Matrix Multiplication Background User's Guide*. 2023. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html.

[35] NVIDIA Corporation. *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/. Accessed: 2025-01-31.

[36] NVIDIA. *NVIDIA Multi-Instance GPU User Guide Documentation*. Tech. rep. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. NVIDIA, Jan. 2025.

[37] Thomas Bradley. *Hyper-Q Sample*. https://developer.download.nvidia.com/compute/DevZone/C/html_x64/6_Advanced/simpleHyperQ/doc/HyperQ.pdf. Accessed: 2025-01-31.

[38] NVIDIA. *Multi-Process Service*. Tech. rep. NVIDIA Corporation, Oct. 2022. URL: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[39] NVIDIA. *NVIDIA Multi-Instance GPU User Guide*. Tech. rep. https://docs.nvidia.com/datacenter/tesla/pdf/NVIDIA_MIG_User_Guide.pdf. NVIDIA, Jan. 2025.

[40] Mahendra Bhatu Gawali and Subhash K Shinde. "Task scheduling and resource allocation in cloud computing using a heuristic approach". In: *Journal of Cloud Computing* 7 (2018), pp. 1–16.

[41] Andy B. Yoo, Morris A. Jette, and Mark Grondona. "SLURM: Simple Linux Utility for Resource Management". In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. 2003, pp. 44–60.

[42] Thomas Bradley. *SLURM Documentation*. https://slurm.schedmd.com/. Accessed: 2022-12-13.

[43] The Kubernetes Authors. *Kubernetes Documentation*. Accessed: 2025-01-27. The Kubernetes Project. n.d. URL: https://kubernetes.io/docs/home/.

[44] Benjamin Hindman et al. “Mesos: A platform for {Fine-Grained} resource sharing in the data center”. In: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. 2011.

[45] Vinod Kumar Vavilapalli et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: Association for Computing Machinery, 2013. ISBN: 9781450324281. DOI: 10.1145/2523616.2523633. URL: https://doi.org/10.1145/2523616.2523633.

[46] Matei Zaharia et al. “Spark: Cluster computing with working sets”. In: *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*. 2010.

[47] IBM. *What Is High-Performance Computing (HPC)?* Accessed: 2025-01-27. n.d. URL: https://www.ibm.com/think/topics/hpc.

[48] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud computing: concepts, technology & architecture*. Pearson Education, 2013.

[49] Qizhen Weng et al. “MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 945–960.

[50] Qinghao Hu et al. “Characterization and prediction of deep learning workloads in large-scale gpu datacenters”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021, pp. 1–15.

[51] Shibo Zheng, Yanzhao Li, Haoning Zhang, et al. “Characterization and Prediction of Deep Learning Workloads in Large-Scale GPU Datacenters”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2021, pp. 1–16.

[52] Zhisheng Ye et al. “Deep Learning Workload Scheduling in GPU Datacenters: A Survey”. In: *ACM Comput. Surv.* 56.6 (Jan. 2024). ISSN: 0360-0300. DOI: 10.1145/3638757. URL: https://doi.org/10.1145/3638757.

[53] Wei Gao et al. “Deep learning workload scheduling in gpu datacenters: Taxonomy, challenges and vision”. In: *arXiv preprint arXiv:2205.11913* (2022).

[54] Wencong Xiao et al. “Gandiva: Introspective Cluster Scheduling for Deep Learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/xiao.

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

[56] Aurick Qiao et al. “Pollux: Co-adaptive cluster scheduling for goodput-optimized deep learning”. In: *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 2021.

[57] Juncheng Gu et al. “Tiresias: A GPU Cluster Manager for Distributed Deep Learning”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019,

pp. 485–500. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/gu.

[58] Abeda Sultana et al. "E-LAS: Design and Analysis of Completion-Time Agnostic Scheduling for Distributed Deep Learning Cluster". In: *49th International Conference on Parallel Processing - ICPP*. ICPP '20. Edmonton, AB, Canada: Association for Computing Machinery, 2020. ISBN: 9781450388160. DOI: 10.1145/3404397.3404415. URL: https://doi.org/10.1145/3404397.3404415.

[59] Hanyu Zhao et al. "HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-19-9.

[60] Xiaorui Zhu et al. "Vapor: A GPU Sharing Scheduler with Communication and Computation Pipeline for Distributed Deep Learning". In: *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. 2021, pp. 108–116. DOI: 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00028.

[61] Gingfung Yeung et al. "Horus: Interference-Aware and Prediction-Based Scheduling in Deep Learning Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 33.1 (2022), pp. 88–100. DOI: 10.1109/TPDS.2021.3079202.

[62] Yizhou Luo et al. "Scheduling Deep Learning Jobs in Multi-Tenant GPU Clusters via Wise Resource Sharing". In: *2024 IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS)*. IEEE. 2024, pp. 1–10.

[63] Kshiteej Mahajan et al. "Themis: Fair and Efficient GPU Cluster Scheduling". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 289–304. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/mahajan.

[64] Tan N. Le et al. "AlloX: Compute Allocation in Hybrid Clusters". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387547. URL: https://doi.org/10.1145/3342195.3387547.

[65] Haoyue Zheng et al. "Cynthia: Cost-Efficient Cloud Resource Provisioning for Predictable Distributed Deep Neural Network Training". In: *Proceedings of the 48th International Conference on Parallel Processing*. ICPP 2019. Kyoto, Japan: Association for Computing Machinery, 2019. ISBN: 9781450362955. DOI: 10.1145/3337821.3337873. URL: https://doi.org/10.1145/3337821.3337873.

[66] Peifeng Yu, Jiachen Liu, and Mosharaf Chowdhury. "Fluid: Resource-aware Hyperparameter Tuning Engine". In: *Proceedings of Machine Learning and Systems*. Ed. by A. Smola, A. Dimakis, and I. Stoica. Vol. 3. 2021, pp. 502–516. URL: %5Chref%7Bhttps://proceedings.mlsys.org/

paper/2021/file/9f61408e3afb633e50cdf1b20de6f466-Paper.pdf%7D.

[67] Deepak Narayanan et al. "Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-19-9.

[68] Laiping Zhao et al. "AITurbo: Unified Compute Allocation for Partial Predictable Training in Commodity Clusters". In: *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '21. Virtual Event, Sweden: Association for Computing Machinery, 2021, pp. 133–145. ISBN: 9781450382175. DOI: 10.1145/3431379.3460639. URL: https://doi.org/10.1145/3431379.3460639.

[69] Yanghua Peng et al. "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190517. URL: https://doi.org/10.1145/3190508.3190517.

[70] Zhenwei Zhang et al. "Prophet: Speeding up Distributed DNN Training with Predictable Communication Scheduling". In: *50th International Conference on Parallel Processing*. ICPP 2021. Lemont, IL, USA: Association for Computing Machinery, 2021. ISBN: 9781450390682. DOI: 10.1145/3472456.3472467. URL: https://doi.org/10.1145/3472456.3472467.

[71] Yanghua Peng et al. "DL2: A Deep Learning-Driven Scheduler for Deep Learning Clusters". In: *IEEE Transactions on Parallel and Distributed Systems* 32.8 (2021), pp. 1947–1960. DOI: 10.1109/TPDS.2021.3052895.

[72] Yixin Bao, Yanghua Peng, and Chuan Wu. "Deep Learning-based Job Placement in Distributed Machine Learning Clusters". In: *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2019, pp. 505–513. DOI: 10.1109/INFOCOM.2019.8737460.

[73] Wencong Xiao et al. "AntMan: Dynamic Scaling on GPU Clusters for Deep Learning". In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. OSDI'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-19-9.

[74] K. R. Jayaram et al. "FfDL: A Flexible Multi-Tenant Deep Learning Platform". In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 82–95. ISBN: 9781450370097. DOI: 10.1145/3361525.3361538. URL: https://doi.org/10.1145/3361525.3361538.

[75] Wencong Xiao et al. "Scheduling CPU for GPU-Based Deep Learning Jobs". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Computing Machinery, 2018, p. 503. ISBN: 9781450360111. DOI: 10.1145/3267809.3275445. URL: https://doi.org/10.1145/3267809.3275445.

[76] Hyungjun Oh et al. "Out-of-Order Backprop: An Effective Scheduling Technique for Deep Learning". In: *Proceedings of the Seventeenth European Conference on Computer Systems*. EuroSys '22. Rennes, France: Association for Computing Machinery, 2022, pp. 435–452. ISBN: 9781450391627.

DOI: 10.1145/3492321.3519563. URL: https://doi.org/10.1145/3492321.3519563.

[77] Jiazhen Gu et al. "DeepProf: Performance Analysis for Deep Learning Applications via Mining GPU Execution Patterns". In: *CoRR* abs/1707.03750 (2017). arXiv: 1707.03750. URL: http://arxiv.org/abs/1707.03750.

[78] Diksha Moolchandani et al. "Performance Prediction for Multi-Application Concurrency on GPUs". In: *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2020, pp. 306–315. DOI: 10.1109/ISPASS48437.2020.00050.

[79] Rachata Ausavarungnirun et al. "MASK: Redesigning the GPU Memory Hierarchy to Support Multi-Application Concurrency". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 503–518. ISBN: 9781450349116.

[80] Minsoo Rhu et al. "Virtualizing Deep Neural Networks for Memory-Efficient Neural Network Design". In: *CoRR* abs/1602.08124 (2016). arXiv: 1602.08124. URL: http://arxiv.org/abs/1602.08124.

[81] Hangchen Yu and Christopher J. Rossbach. "Full Virtualization for GPUs Reconsidered". In: 2017.

[82] Sebastian Baunsgaard, Sebastian B. Wrede, and Pınar Tozun. *Training for Speech Recognition on Coprocessors*. 2020. arXiv: 2003.12366 [eess.AS].

[83] Cheng Tan et al. "Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem". In: *CoRR* abs/2109.11067 (2021). arXiv: 2109.11067. URL: https://arxiv.org/abs/2109.11067.

[84] Priya Goyal et al. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *CoRR* abs/1706.02677 (2017). arXiv: 1706.02677. URL: http://arxiv.org/abs/1706.02677.

[85] Dror G. Feitelson. "Packing Schemes for Gang Scheduling". In: *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*. IPPS '96. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 89–110. ISBN: 3540618643.

[86] Alexandros Koliousis et al. "Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers". In: *PVLDB* 12.11 (2019), pp. 1399–1412.

[87] Shang Wang et al. "Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models". In: *CoRR* abs/2102.02344 (2021). arXiv: 2102.02344. URL: https://arxiv.org/abs/2102.02344.

[88] Haichen Shen et al. "Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. SOSP '19. Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 322–337. ISBN: 9781450368735. DOI: 10.1145/3341301.3359658. URL: https://doi.org/10.1145/3341301.3359658.

[89] Andrew Or, Haoyu Zhang, and Michael Freedman. "Resource Elasticity in Distributed Deep Learning". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 400–411.

[90] Deepak Narayanan et al. "Accelerating Deep Learning Workloads Through Efficient Multi-Model Execution". In: *NeurIPS Workshop on Systems for Machine Learning*. Dec. 2018. URL: https://www.microsoft.com/en-us/research/publication/accelerating-deep-learning-workloads-through-efficient-multi-model-execution/.

[91] David A. Patterson et al. "Carbon Emissions and Large Neural Network Training". In: *CoRR* abs/2104.10350 (2021). arXiv: 2104.10350. URL: https://arxiv.org/abs/2104.10350.

[92] Myeongjae Jeon et al. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, pp. 947–960.

[93] Shang Wang et al. "Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models". In: *MLSys* 3 (2021), pp. 599–623.

[94] Anastassia Ailamaki et al. "DBMSs on a Modern Processor: Where Does Time Go?" In: *VLDB*. 1999, pp. 266–277.

[95] Michael Ferdman et al. "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware". In: *ASPLOS*. 2012, pp. 37–48.

[96] Pınar Tözün et al. "From A to E: Analyzing TPC's OLTP Benchmarks: The Obsolete, the Ubiquitous, the Unexplored". In: *EDBT*. 2013, pp. 17–28.

[97] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. "A methodology for OLTP micro-architectural analysis". In: *DaMoN @ ACM SIGMOD*. 2017, 1:1–1:10.

[98] Svilen Kanev et al. "Profiling a Warehouse-Scale Computer". In: *ISCA*. 2015, pp. 158–169.

[99] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *ASPLOS*. 2014, pp. 127–144.

[100] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. "OLTP in Wonderland: Where Do Cache Misses Come from in Major OLTP Components?" In: *DaMoN @ ACM SIGMOD*. 2013, pp. 1–6.

[101] Kimberly Keeton et al. "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads". In: *ISCA*. 1998, pp. 15–26.

[102] *TensorBoard: TensorFlow's visualization toolkit*. https://www.tensorflow.org/tensorboard.

[103] *PyTorch Profiler*. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.

[104] NVIDIA Corporation. *NVIDIA Nsight Systems*. https://developer.nvidia.com/nsight-systems. Accessed: 2022-12-21.

[105] *NVTX (NVIDIA Tools Extension Library)*. https://nvidia.github.io/NVTX/.

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

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

[108] *NVIDIA Management Library (NVML)*. https://developer.nvidia.com/nvidia-management-library-nvml.

[109] *NVIDIA Data Center GPU Manager*. https://github.com/NVIDIA/DCGM.

[110] Bryan McCann and contributors. *PyTorch Example*. https://github.com/pytorch/examples/tree/main/mnist. 2022.

[111] Li Deng. "The mnist database of handwritten digit images for machine learning research". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 141–142.

[112] Kaiming He et al. "Deep residual learning for image recognition". In: *CVPR*. 2016, pp. 770–778.

[113] Ross Wightman. *PyTorch Image Models*. https://github.com/rwightman/pytorch-image-models. 2019.

[114] *top(1) — Linux manual page*. https://man7.org/linux/man-pages/man1/top.1.html.

[115] *ROCm Documentation*. https://sep5.readthedocs.io/en/latest/ROCm_System_Managment/ROCm-System-Managment.html.

[116] *NVTOP: Neat Videocard TOP*. https://github.com/Syllo/nvtop.

[117] Pouya Kousha et al. "Designing a Profiling and Visualization Tool for Scalable and In-depth Analysis of High-Performance GPU Clusters". In: *IEEE HiPC*. 2019, pp. 93–102.

[118] NVIDIA. *CUDA Profiling Tools Interface (CUPTI)*. https://docs.nvidia.com/cupti/.

[119] Laksono Adhianto et al. "HPCToolkit: Tools for performance analysis of optimized parallel programs". In: *Concurrency and Computation: Practice and Experience* 22.6 (2010), pp. 685–701.

[120] Sameer S Shende and Allen D Malony. "The TAU parallel performance system". In: *IJHPCA* 20.2 (2006), pp. 287–311.

[121] Hui Zhang and Jeffrey Hollingsworth. "Understanding the performance of GPGPU applications from a data-centric view". In: *ProTools*. 2019, pp. 1–8.

[122] Keren Zhou, Mark Krentel, and John Mellor-Crummey. "A Tool for Top-down Performance Analysis of GPU-Accelerated Applications". In: *PPoPP*. 2020, pp. 415–416.

[123] Yuting Jiang et al. "Moneo: Monitoring Fine-Grained Metrics Nonintrusively in AI Infrastructure". In: *ACM SIGOPS Oper. Syst. Rev.* 56.1 (2022), pp. 18–25.

[124] Hari Subramoni et al. "INAM2: InfiniBand network analysis and monitoring with MPI". In: *High Performance Computing*. 2016, pp. 300–320.

[125] Qizhen Weng et al. "MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters". In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Apr. 2022, pp. 945–960. URL: https://www.usenix.org/conference/nsdi22/presentation/weng.

[126] Konstantinos Nikas et al. “DICER: Diligent Cache Partitioning for Efficient Workload Consolidation”. In: *ICPP*. 2019.

[127] Benjamin Hindman et al. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. In: *NSDI*. 2011, pp. 295–308.

[128] *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/. Accessed: 2022-10-21.

[129] *NVIDIA Multi-Instance GPU User Guide*. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/. NVIDIA. Aug. 2021.

[130] Mehmet E. Belviranli et al. “CuMAS: Data Transfer Aware Multi-Application Scheduling for Shared GPUs”. In: *Proceedings of the 2016 International Conference on Supercomputing*. ICS ’16. Association for Computing Machinery, 2016.

[131] Vignesh T. Ravi et al. “Supporting GPU Sharing in Cloud Environments with a Transparent Runtime Consolidation Framework”. In: HPDC ’11. Association for Computing Machinery, 2011, pp. 217–228.

[132] Sina Darabi et al. “NURA: A Framework for Supporting Non-Uniform Resource Accesses in GPUs”. In: *Proc. ACM Meas. Anal. Comput. Syst.* 6.1 (Feb. 2022).

[133] Hongwen Dai et al. “Accelerate GPU Concurrent Kernel Execution by Mitigating Memory Pipeline Stalls”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018, pp. 208–220.

[134] Zhenning Wang et al. “Simultaneous Multikernel GPU: Multi-tasking throughput processors via fine-grained sharing”. In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016, pp. 358–369.

[135] Zhenning Wang et al. “Quality of service support for fine-grained sharing on GPUs”. In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 269–281.

[136] Qiumin Xu et al. “Warped-Slicer: Efficient Intra-SM Slicing through Dynamic Resource Partitioning for GPU Multiprogramming”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 230–242.

[137] Xia Zhao, Zhiying Wang, and Lieven Eeckhout. “Classification-Driven Search for Effective SM Partitioning in Multitasking GPUs”. In: *Proceedings of the 2018 International Conference on Supercomputing*. 2018, pp. 65–75.

[138] Baolin Li et al. “MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters”. In: *ACM SoCC*. 2022.

[139] Baolin Li et al. “Characterizing Multi-Instance GPU for Machine Learning Workloads”. In: *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2022, pp. 724–731.

[140] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. University of Toronto, 2009.

[141] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. “A Downsampled Variant of ImageNet as an Alternative to the CIFAR datasets”. In: *CoRR arXiv* (2017).

[142] Criteo. *Criteo 1TB Click Logs dataset*. https://www.criteo.com/news/press-releases/2015/07/criteo-releases-industrys-largest-ever-dataset/.

[143] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CVPR*. 2016, pp. 770–778.

[144] Mingxing Tan and Quoc V. Le. "EfficientNetV2: Smaller Models and Faster Training". In: *CoRR* abs/2104.00298 (2021). arXiv: 2104.00298. URL: https://arxiv.org/abs/2104.00298.

[145] Hugo Touvron et al. "Going deeper with image transformers". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, pp. 32–42.

[146] Maxim Naumov et al. "Deep Learning Recommendation Model for Personalization and Recommendation Systems". In: *CoRR* abs/1906.00091 (2019). URL: https://arxiv.org/abs/1906.00091.

[147] Ties Robroek et al. "Data Management and Visualization for Benchmarking Deep Learning Training Systems". In: *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning, DEEM 2023, Seattle, WA, USA, 18 June 2023*. ACM, 2023, 1:1–1:5. DOI: 10.1145/3595360.3595851. URL: https://doi.org/10.1145/3595360.3595851.

[148] NVIDIA. *Data Center GPU Manager Documentation*. Tech. rep. https://docs.nvidia.com/datacenter/dcgm/latest/dcgm-user-guide/. NVIDIA, Mar. 2022.

[149] PyTorch Contributors. *Fake Tensor Mode in PyTorch*. Accessed: 2023-10-31. 2023. URL: https://pytorch.org/docs/stable/torch.compiler_fake_tensor.html.

[150] DeepSpeed. *Memory Requirements*. Accessed: 2025-01-31. 2025. URL: https://deepspeed.readthedocs.io/en/latest/memory.html.

[151] PyTorch Developers. *PyTorch Autograd Documentation*. Accessed: January 24, 2025. 2024. URL: https://pytorch.org/docs/stable/notes/autograd.html.

[152] Cong Guo et al. "GMLake: Efficient and Transparent GPU Memory Defragmentation for Large-scale DNN Training with Virtual Memory Stitching". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2024, pp. 450–466.

[153] Tianqi Chen et al. "Training Deep Nets with Sublinear Memory Cost". In: 2016.

[154] Jianwei Feng and Dong Huang. "Optimal gradient checkpoint search for arbitrary computation graphs". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 11433–11442.

[155] NVIDIA. *Train With Mixed Precision*. Accessed: 2025-01-31. 2025. URL: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html.

[156] P Kingma Diederik. "Adam: A method for stochastic optimization". In: *(No Title)* (2014).

[157] Geoffrey Hinton. *Neural Networks for Machine Learning Lecture 6a Overview of mini-batch gradient descent*. Coursera Lecture. 2012. URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[158] Taeyoon Kim et al. "FusionFlow: Accelerating Data Preprocessing for Machine Learning with CPU-GPU Cooperation". In: *Proceedings of the VLDB Endowment* 17.4 (2023), pp. 863–876.

[159] Dong Xu et al. "Efficient Tensor Offloading for Large Deep-Learning Model Training based on Compute Express Link". In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2024, pp. 1–18.

[160] Terrence J Sejnowski and Charles R Rosenberg. "Parallel networks that learn to pronounce English text". In: *Complex systems* 1.1 (1987), pp. 145–168.

[161] Yanping Huang et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism". In: (2019).

[162] Deepak Narayanan et al. "PipeDream: Generalized pipeline parallelism for DNN training". In: *Proceedings of the 27th ACM symposium on operating systems principles*. 2019, pp. 1–15.

[163] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond data and model parallelism for deep neural networks." In: *Proceedings of Machine Learning and Systems* 1 (2019), pp. 1–13.

[164] NVIDIA Corporation. *NVIDIA Tesla V100 GPU Architecture*. Accessed: 2025-02-05. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[165] NVIDIA. *NVIDIA Ampere Architecture Whitepaper*. 2020. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[166] NVIDIA Developer Forums. *Same program Different memory usage on Different GPU*. Accessed: 2025-02-05. 2022. URL: https://forums.developer.nvidia.com/t/same-program-different-memory-usage-on-different-gpu/210096.

[167] Taeho Kim et al. "LLMem: Estimating GPU Memory Usage for Fine-Tuning Pre-Trained LLMs". In: *arXiv preprint arXiv:2404.10933* (2024).

[168] Yanjie Gao et al. "Estimating GPU memory consumption of deep learning models". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, pp. 1342–1352.

[169] Yanjie Gao et al. "Runtime performance prediction for deep learning models with graph neural network". In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2023, pp. 368–380.

[170] mmarouen. *BUG Deepspeed memory allocation estimation different than real!* https://github.com/deepspeedai/DeepSpeed/issues/5484. Accessed: 2025-02-06. 2024.

[171] Aaron Shi and Zachary DeVito. *Understanding GPU Memory 1: Visualizing All Allocations over Time*. Accessed: 2025-01-31. Dec. 2023. URL: https://pytorch.org/blog/understanding-gpu-memory-1/.

[172] Shankar Pandala. *Lazy Predict: A tool to quickly build machine learning models without coding*. Accessed: 2024-09-07. 2020. URL: https://github.com/shankarpandala/lazypredict.

[173] Pierre Geurts, Damien Ernst, and Louis Wehenkel. "Extremely randomized trees". In: *Machine learning* 63 (2006), pp. 3–42.

[174] Guolin Ke et al. "Lightgbm: A highly efficient gradient boosting decision tree". In: *Advances in neural information processing systems* 30 (2017).

[175] Leo Breiman. "Random forests". In: *Machine learning* 45 (2001), pp. 5–32.

[176] Tianqi Chen and Carlos Guestrin. "Xgboost: A scalable tree boosting system". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[177] Carolin Strobl et al. "Conditional variable importance for random forests". In: *BMC bioinformatics* 9 (2008), pp. 1–11.

[178] Chao Chen, Andy Liaw, Leo Breiman, et al. "Using random forest to learn imbalanced data". In: *University of California, Berkeley* 110.1-12 (2004), p. 24.

[179] Laurens Van der Maaten and Geoffrey Hinton. "Visualizing data using t-SNE." In: *Journal of machine learning research* 9.11 (2008).

[180] Christopher M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[181] David Opitz and Richard Maclin. "Popular ensemble methods: An empirical study". In: *Journal of artificial intelligence research* 11 (1999), pp. 169–198.

[182] Thomas G Dietterich. "Ensemble methods in machine learning". In: *International workshop on multiple classifier systems*. Springer. 2000, pp. 1–15.

[183] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. CRC press, 2025.

[184] Geoffrey E Hinton and Richard Zemel. "Autoencoders, minimum description length and Helmholtz free energy". In: *Advances in neural information processing systems* 6 (1993).

[185] Nitish Srivastava et al. "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958.

[186] Sergey Ioffe. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[187] Alina Kuznetsova et al. "The Open Images Dataset V4". In: *International Journal of Computer Vision* 128 (2018), pp. 1956–1981. URL: https://api.semanticscholar.org/CorpusID:53296866.

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

[189] Alejandro Newell, Kaiyu Yang, and Jia Deng. "Stacked Hourglass Networks for Human Pose Estimation". In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 483–499. ISBN: 978-3-319-46484-8.

[190] Gao Huang et al. “Densely Connected Convolutional Networks”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, pp. 2261–2269. DOI: 10.1109/CVPR.2017.243.

[191] Yanjie Gao et al. “An Empirical Study on Low GPU Utilization of Deep Learning Jobs”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.

[192] Wencong Xiao et al. “Gandiva: Introspective Cluster Scheduling for Deep Learning”. In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/xiao.

[193] Connor Espenshade et al. “Characterizing Training Performance and Energy for Foundation Models and Image Classifiers on Multi-Instance GPUs”. In: *Proceedings of the 4th Workshop on Machine Learning and Systems*. 2024, pp. 47–55.

[194] Foteini Strati, Xianzhe Ma, and Ana Klimovic. “Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications”. In: *Proceedings of the Nineteenth European Conference on Computer Systems*. 2024, pp. 1075–1092.

[195] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[196] Zhisheng Ye et al. “ASTRAEA: A Fair Deep Learning Scheduler for Multi-Tenant GPU Clusters”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.11 (2022), pp. 2781–2793. DOI: 10.1109/TPDS.2021.3136245.

[197] Microsoft Research. *Philly Traces*. Accessed: 2025-02-12. 2020. URL: https://github.com/msr-fiddle/philly-traces.