Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 505

Systems Group, Department of Computer Science, ETH Zurich

Towards resource and interference-aware scheduling of ML workloads

by

Paul Elvinger

Supervised by

Prof. Ana Klimovic, Foteini Strati

March 2024 – September 2024

Abstract

Graphics Processing Units (GPUs) are crucial for Deep Neural Network (DNN) operations, yet they often suffer from under-utilization due to over-allocation or workloads unable to fully exploit their capabilities. To address this, spatial sharing techniques have emerged, allowing multiple workloads to share GPUs. However, model colocation can lead to interference, increasing inference latencies and potentially violating Service Level Objectives (SLOs) in ML serving scenarios. While various ML inference cluster schedulers have proposed different colocation approaches, comparing their performance has been challenging due to diverse underlying architectures and metrics for modeling GPU utilization and interference. To overcome this, we present a novel modular ML inference cluster framework that abstracts core scheduler components and enables easy integration of various colocation policies. We implemented and evaluated state-of-the-art ML inference colocation policies within this framework, allowing for a comprehensive analysis under various SLOs and loads. Our evaluations reveal that effective ML inference cluster schedulers must accurately model interference at multiple levels, optimize batch sizes for inference, and account for intra-cluster queuing and data transmission delays. Furthermore, we introduce fine-grained profiling measurements at the kernel granularity, defining new metrics to quantify model resource requirements more precisely. This work contributes to advancing ML inference scheduling techniques by providing a common ground for comparison, identifying crucial properties for schedulers and laying the foundations for more efficient resource allocation strategies in GPU clusters.

Acknowledgements

I would like to express my deepest gratitude to Foteini Strati for her invaluable guidance and support throughout the course of this thesis. Over the past 12 months, I have had the privilege of collaborating with Foteini on several projects, including this one, which has provided me with profound insights into the world of research. Her talent and problem-solving approach have been a true source of inspiration. I am especially grateful for the time she dedicated to our discussions, often during extended meetings. These conversations have been crucial to the progress and content of this work.

I am also sincerely grateful to Prof. Ana Klimovic for giving me the opportunity to work on these projects and for her continued guidance and involvement. Her insightful feedback and direction have been of great value for this work. I have particularly appreciated how she and the entire EASL group, have welcomed me as a part of the research group. Throughout my 12-month involvement, I felt like an integral member of the group rather than just a student.

The active roles both Foteini and Ana have played in this thesis, through their dedication of time and effort to discussions and feedback, have been invaluable and deeply appreciated. Their involvement has made this work so much more interesting and engaging.

To Foteini and Ana, thank you both so much for your time, expertise, and unwavering support.

Contents

Declaration

The author of this thesis must acknowledge the use of generative artificial intelligence for different parts of this work.

- **Text Revision and Spell-Checking**: Claude 3.5 Sonnet [1] was utilized to revise the text for conciseness and spell-checking. The entire report is based on the author's own words.
- **Debugging Assistance**: Claude 3.5 Sonnet [1] and occasional use of GPT-4o [35] were utilized on multiple occasions throughout this work for debugging purposes.
- **Plotting Script Assistance**: GitHub Copilot [9] was used as an assistance for writing the plotting scripts.

Chapter 1

Introduction

Deep Neural Networks (DNNs) have emerged as the driving force behind numerous remarkable advancements in machine learning (ML) and artificial intelligence (AI) in recent years. This success can be largely attributed to innovations such as Convolutional Neural Networks (CNNs) [69] and, more recently, Large Language Models (LLMs) [1, 12, 34] built on transformer architectures [78]. While initially designed for gaming applications, Graphics Processing Units (GPUs) have become indispensable for ML workloads due to their capacity for large-scale parallel computations. Due to their high computational capabilities, several studies [65, 80] have revealed that GPUs are often underutilized, leading to inefficiencies in resource usage. Given the high costs and limited availability of GPUs, there has been a growing interest within the research community to improve the utilization of these critical hardware accelerators.

One popular technique to increase GPU utilization was the use of larger batch sizes, which not only improves throughput but also ensures more efficient resource use. While this approach is suitable for training workloads that may run for days or months, it proves inadequate for inference workloads, which are typically completed in milliseconds or seconds. Larger batch sizes, although efficient in utilizing resources, can lead to higher latencies for individual requests-an undesirable outcome for inference tasks that are often part of latency-critical applications subject to strict Service Level Objectives (SLOs). Consequently, the batch sizes used in inference are constrained by these SLOs, limiting their effectiveness. Moreover, as GPUs become increasingly powerful, it is becoming challenging for individual applications to fully utilize and saturate these resources.

To address this challenge, both industry and academia have shifted towards enabling the sharing of GPUs across multiple inference workloads. The two main strategies for sharing GPUs are temporal sharing and spatial sharing. Temporal sharing involves dividing GPU time among different workloads, adjusting the time slices based on workload demand [66, 82].

However, this method often fails to fully utilize the GPU's resources since individual workloads may still leave parts of the GPU underutilized. As a result, attention has turned toward spatial sharing, where multiple workloads can execute concurrently by partitioning the GPU's resources.

Hardware manufacturers, such as NVIDIA, have introduced hardware-level features [23, 24] to support spatial sharing, and recent research has built extensively on these innovations. Several studies have proposed different approaches for spatially sharing GPUs to minimize interference between colocated models and ensure that requests are served within their SLOs. These approaches typically involve estimating a model's resource requirements to determine which models can be colocated without exceeding the GPU's capacity while still guaranteeing SLOs. Common techniques include profiling models upfront to measure their exact resource requirements when colocated with other models [60], training machine learning-based resource estimators [72, 79], or developing models to predict interference [72, 81].

However, we observe two key characteristics of state-of-the-art systems that limit their comparability and effectiveness. Each system models GPU utilization and interference in its own unique way, raising questions about which metrics are most accurate and at what granularity they should be applied. Furthermore, these systems are often based on different underlying frameworks, frequently tightly coupled to specific use cases or unnecessarily complex (e.g., Triton [31], Ray Serve [51]). As a consequence, comparing different approaches is very challenging, as there is no common ground where the underlying system remains constant while only the model placement or colocation policy varies. This lack of standardization makes it hard to identify the most effective strategies for ML inference scheduling and resource allocation in GPU clusters.

To address these issues, this thesis makes the following contributions:

1. We introduce a novel modular serving system that provides the core architecture for any ML inference cluster scheduler and allows for easy exchange of various components. This system abstracts core components of existing ML inference cluster schedulers, enabling the integration and evaluation of different placement policies on common grounds.

2. Using this system, we have implemented and evaluated state-of-the-art ML inference colocation policies, conducting a comprehensive performance analysis under various Service Level Objectives and loads. This approach enables us to identify several shortcomings and suggest modifications to circumvent these limitations.

3. Based on this extensive analysis, we uncover key insights that should drive ML inference colocation policies in future work, such as:

    - The clear need to accurately model interference at various levels, a task that is only partially achieved by related work.

- The importance of modeling intra-cluster queuing and data transmission delays. Related work is mostly limited to modeling what happens at the level of the GPU.

4. We conduct fine-grained profiling measurements at the kernel granularity, allowing us to define new metrics to quantify model resource requirements.

The remainder of this work is structured as follows: Chapter 2 introduces the necessary background and lays the foundation for this work. Chapter 3 summarizes the most relevant recent work in the space of ML inference schedulers. Chapter 4 introduces and explains our novel modular ML inference scheduler in great detail. In Chapter 5, we explain the placement policies that will be evaluated as part of this work. We present the corresponding results and analysis in Chapter 6, and we conclude our work with Chapter 7, where we summarize our findings and discuss future work.

Chapter 2

Background

This section provides the essential background information necessary for understanding the remainder of this work. We begin by analyzing the key differences between Machine Learning training and inference in Section 2.1. Section 2.2 offers a comprehensive overview of GPU architecture. In Section 2.3, we delve into various GPU sharing techniques, exploring methods for optimizing resource utilization. Finally, Section 2.4 presents an overview of existing ML inference systems.

2.1 Machine Learning Training vs. Inference

Machine Learning (ML) models undergo two primary phases during their lifecycle: training and inference. These phases differ significantly in their objectives, operational constraints, and resource requirements.

The training phase involves iterative processing of large datasets to learn a function that maps inputs to outputs. During this phase, models perform successive forward-backward passes over the training data, applying a function to inputs and updating the model parameters with respect to a loss function. This process is computationally intensive, with state-of-the-art models often requiring months to reach high levels of accuracy [1, 12, 34]. Training typically occurs offline and is optimized for throughput, maximizing the number of processed samples per unit of time.

In contrast, the inference phase applies the trained model to new data for prediction, involving only a forward pass. Inference often operates under strict real-time constraints, with latencies measured in milliseconds or seconds. Many ML applications crucial to systems such as search engines, advertising platforms, and autonomous vehicles [67, 73] must meet specific Service Level Objectives (SLOs) for response times. This requirement necessitates that responses be available within a specified timeframe to be considered valid. Unlike training, inference usually occurs online, with data arriving at irregular intervals as part of requests.

The distinct nature of these phases leads to divergent resource allocation strategies. While training is primarily optimized for throughput to accelerate the overall process, inference requires a balanced optimization between throughput and individual request latency. Data availability also differs significantly: training data is typically available upfront, allowing for more predictable resource allocation, whereas inference data arrives in real-time, necessitating more flexible resource management to account for fluctuating loads.

The increasing complexity and size of ML models have necessitated a shift from commodity hardware to specialized computing clusters. Modern ML systems often rely on hardware accelerators such as GPUs or TPUs [14] to offload the heavy computation of model training and inference. Given the high operational costs of these clusters, it is crucial to implement effective scheduling and resource allocation strategies that balance performance, efficiency, and cost-effectiveness.

2.2 Graphics Processing Units (GPUs)

Modern ML training and inference are primarily powered by specialized hardware accelerators that provide huge amount of computational resources in order to exploit parallel programming patterns in program execution. Among these, Graphics Processing Units (GPUs) have gained significant attention in recent years. While other hardware accelerators, such as Google's Tensor Processing Units (TPUs), can serve similar purposes, this work focuses exclusively on GPUs. Furthermore, given NVIDIA's market dominance in the GPU sector we have decided to exclusively use NVIDIA GPUs as part of this work. As a consequence this section will concentrate on NVIDIA GPUs, providing a high-level overview of their programming model and architecture.

2.2.1 NVIDIA Programming Model

The typical process for leveraging GPUs for computation involves three main steps:

1. The data and the program need to be transferred from the host (CPU) to the device (GPU).
2. Next the program is executed on the device.
3. Finally the execution results are copied from the device back to the host.

To execute code on the GPU, programmers must write their programs in the form of CUDA kernels. CUDA [6], which stands for Compute Unified

Device Architecture, is an extension of C/C++ programming and serves as NVIDIA's parallel computing platform and API for GPU programming.

Once the program and data are transferred to the GPU, CUDA kernels are executed by a collection of GPU *threads*. These threads are organized into *thread blocks*, which are further organized into a *grid*. Threads within a block typically collaborate in parallel to compute a value. Thread blocks and grids can be two- or three-dimensional, and it is up to the programmer to specify their dimension upon kernel launch, determining the total number of threads used for the kernel execution.

Within a block, threads are further grouped into warps, each consisting of 32 threads. A warp constitutes the unit of scheduling in each clock cycle, meaning instructions are always dispatched to groups of 32 threads in parallel (Single Instruction Multiple Threads, SIMT). The number of warps that can run concurrently depends on the compute capabilities of individual GPUs.

It is important to note that GPU threads differ significantly from CPU threads. CPU threads are generally heavyweight entities that must be swapped on and off the CPU to provide multi-threaded capabilities, a process that is slow and expensive compared to GPU threading. In contrast, GPU threads are extremely lightweight, with hundreds to thousands typically queued up for work. Since separate registers are allocated to all active threads, no swapping of registers or other state is necessary when switching among GPU threads. GPU resources remain allocated to each thread until it completes its execution.

2.2.2 NVIDIA GPU architecture

Figure 2.1 provides a simplified view of the NVIDIA GPU architecture, often referred to as the *device*.

At the core of NVIDIA GPUs are Streaming Multiprocessors (SMs or SMXs). Modern NVIDIA GPUs typically contain tens to hundreds of these SMs. Each SM comprises a large set of functional units such as FP32, FP64, or Tensor Cores, each supporting and particularly suited for different types of instructions. For instance, tensor cores are specifically designed for efficient matrix multiplication operations.

In addition to functional units, each SM hosts a substantial number of registers that can be used by different threads during execution, as well as some amount of shared memory and an L1 cache. Each thread is allocated a specific number of registers from the register file for its execution. These registers are private to the thread, and threads cannot access each other's registers.

Shared memory provides a fast mechanism for inter-thread communication and data sharing between threads within the same block, offering an alternative to relying on global memory. It is typically not used unless ex-

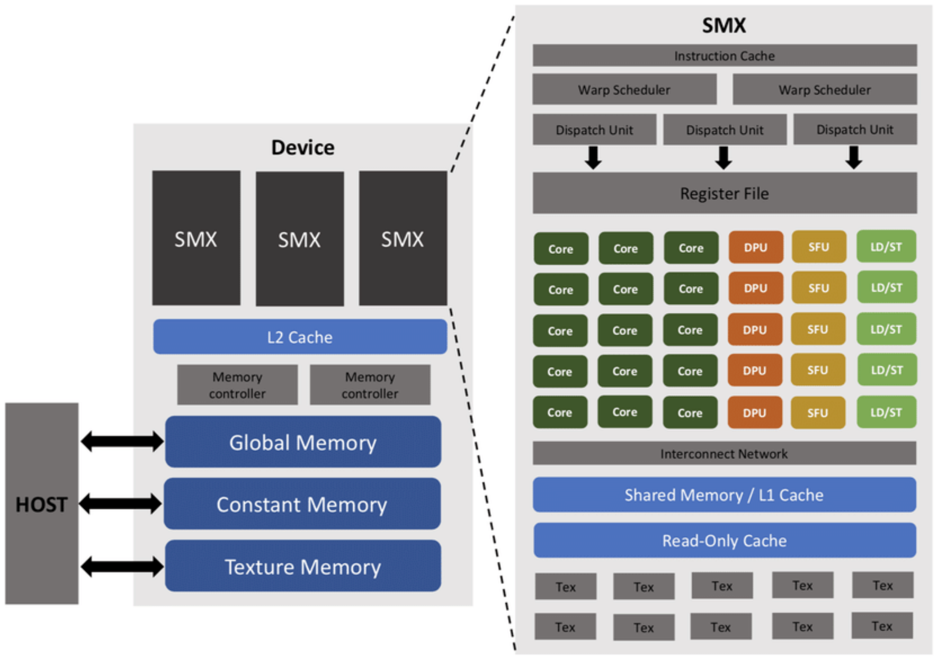


Figure 2.1: Simplified architecture of an NVIDIA GPU. The figure was copied from Afra et al. [59]

plicitly allocated by the programmer. Shared memory is on-chip and shares space with the L1 cache, which is used to cache access to the L2 and global memory. There is a configurable split that defines how much space is used for the L1 cache and shared memory. Unlike shared memory, the L1 cache is transparent to the programmer and cannot be accessed programmatically. It typically has a size of 64 to 128KB per SM.

Below the L1 cache and shared memory in the memory hierarchy is the L2 cache. The L2 cache is still on-chip and is shared among all SMs. It is used to cache accesses to global memory and typically ranges in size from 512KB to a few MB.

At the bottom of the hierarchy is the global memory, also referred to as *Device Memory*. The device memory is no longer on-chip and can be considered the GPU equivalent of RAM on the CPU. It provides the largest memory space but is also the slowest to access on the GPU. Data stored in global memory persists across kernel invocations and is explicitly managed by the programmer.

Other types of memory such as Constant Memory or Texture Memory exist, but these go beyond the scope of knowledge required for this work.

NVIDIA provides a set of profilers that allow programmers to target specific components and levels of the GPU architecture and memory hierarchy. Notably, the NVIDIA Nsight Systems (*nsys*) profiler [28] is designed for gathering system-wide information, providing a global overview of an entire program's performance. The NVIDIA Nsight Compute (*ncu*) profiler [26] allows for the collection of very detailed metrics at the granularity of individual kernels. The System Manager Interface (`nvidia-smi`) profiler [29] can be used to monitor entire GPU devices. While additional profilers like NVIDIA Nsight Graphics [27] exist, they are not relevant in the context of this work.

2.3 GPU Sharing Techniques

The high operational and environmental costs associated with running clusters of GPUs [70] necessitate maximizing their utilization and efficiency. While increasing batch sizes has been a popular method to improve GPU utilization, this approach is limited in the context of ML inference due to the strict Service Level Objectives (SLOs) that must be met for individual requests. Furthermore, as newer GPUs become increasingly powerful with impressive parallel computation capabilities, individual programs often struggle to fully exploit these resources. Consequently, research has shifted towards exploring GPU sharing among multiple workloads. This section introduces two main techniques for GPU sharing: temporal and spatial sharing.

2.3.1 Temporal Sharing

Temporal sharing of GPUs involves allocating GPU resources to different tasks over time, rather than simultaneously. In this model, workloads are scheduled to take turns accessing the GPU, preventing parallel execution. This technique can be particularly useful when models sharing the GPU have different arrival patterns. Time slots can be allocated to each model based on their arrival distribution, ensuring that models experiencing higher load receive more GPU time. However, due to the small batch sizes typically used for ML inference, individual jobs often cannot fully exploit the GPU resources using this method alone and utilization remains poor.

2.3.2 Spatial Sharing

As an alternative to temporal sharing, research has begun to explore spatial sharing, which allows multiple workloads to run in parallel without waiting for allocated time slots, thereby exploiting underutilized resources. However, as multiple applications execute on the same GPU, they may interfere with each other, potentially resulting in higher latencies which is critical for ML inference workloads. This section provides an overview of NVIDIA's hardware support for spatial colocation.

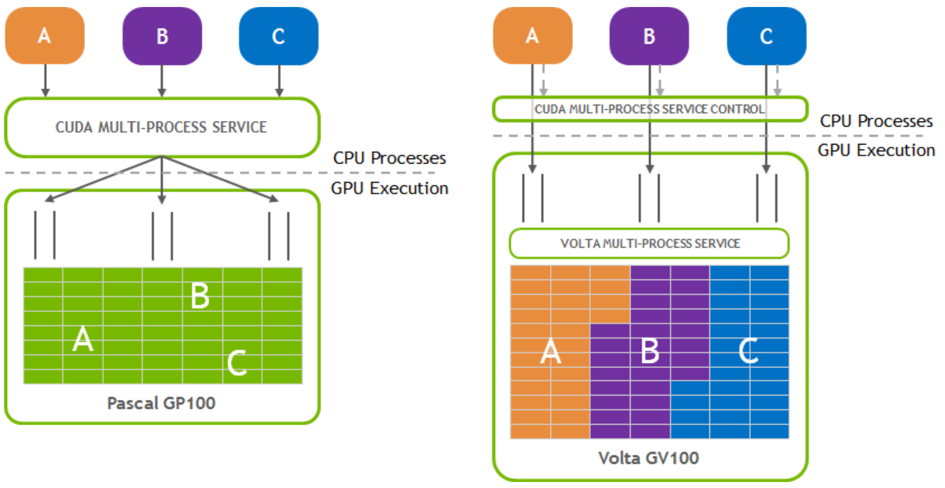

A
B
C
CUDA MULTI-PROCESS SERVICE
CPU Processes
GPU Execution
A
B
C
Pascal GP100
A
B
C
CUDA MULTI-PROCESS SERVICE CONTROL
CPU Processes
GPU Execution
VOLTA MULTI-PROCESS SERVICE
A
B
C
Volta GV100

Figure 2.2: NVIDIA Multi-Process Service (MPS): pre-Volta vs post-Volta. The figure was taken from the official NVIDIA MPS documentation [24].

NVIDIA Multi-Process Service (MPS)

For GPUs with a compute capability ≥ 3.5, NVIDIA introduced the Multi-Process Service (MPS) [24] hardware support, enabling multiple CUDA applications to run simultaneously on the same GPU by sharing its resources.

When a CUDA program initiates, it creates a *CUDA context* that encapsulates all the hardware resources necessary for the program to manage memory and launch work on the GPU. By default, without MPS, CUDA does not support concurrent execution of workloads belonging to different CUDA contexts. Instead, these workloads are scheduled in a time-sliced manner.

In pre-Volta GPU architectures, activating MPS involves running an MPS server in the background. This server consolidates potentially multiple CUDA contexts into a single CUDA context, making it appear as if all work is part of the same program. Consequently, concurrent execution becomes possible, and GPU resources are shared among all workloads.

From Volta GPU architectures onward, NVIDIA introduced additional support allowing to further constrain applications to separate sets of Streaming Multiprocessors (SMs). It is important to note that lower-level resources, such as the L2 cache, are still shared among all applications. In this case, applications bypass the MPS server and submit their workload directly to the hardware. The MPS server remains responsible for ensuring simultaneous scheduling among the shared resources. Figure 2.2 illustrates the difference between pre-Volta and post-Volta MPS implementations. It should be noted that constraining an application to a set of SMs is optional. It is still possible in post-Volta architectures to run MPS without restricting the set of available SMs to a program.

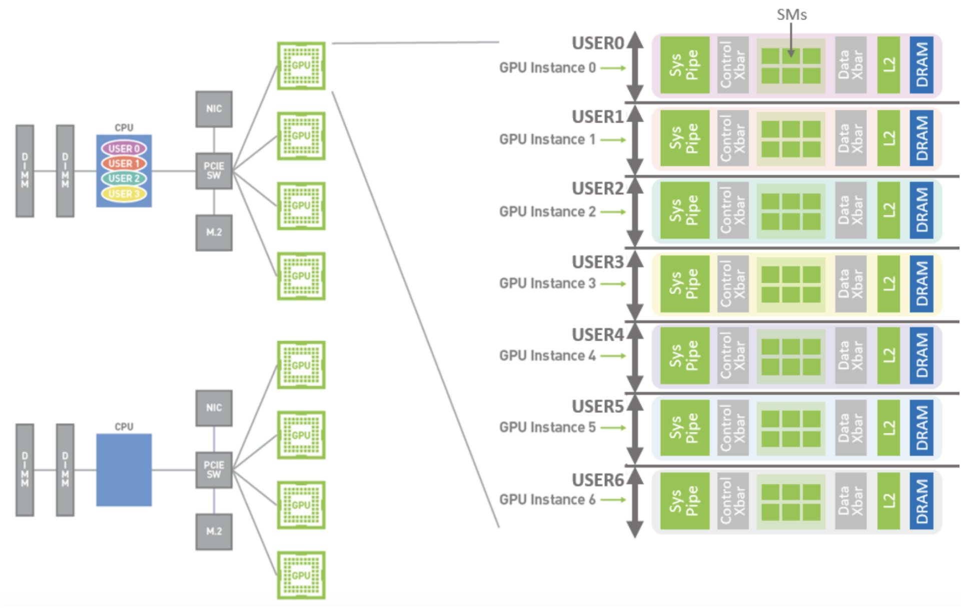


Figure 2.3: NVIDIA Multi Instance GPU (MIG). The figure was taken from the official NVIDIA MIG documentation [23].

It is crucial to understand that MPS does not provide error isolation between different CUDA applications. Consequently, an error in one application may result in the failure of all other CUDA applications as well.

MPS should not be confused with CUDA streams. While MPS utilizes CUDA streams underneath, its purpose is to enable simultaneous execution of different CUDA applications running as separate processes. In contrast, CUDA streams are used as a means to overlap independent computations within the same CUDA application through the use of multiple streams.

NVIDIA Multi-Process Instance (MIG)

NVIDIA introduced Multi-Instance GPU (MIG) [23] technology with the Ampere architecture, requiring a compute capability of 8.0 or higher. MIG represents a significant advancement in GPU virtualization and resource partitioning, offering a more robust and secure approach to GPU sharing. Figure 2.3 illustrates a high-level overview of MIG.

MIG allows for the secure partitioning of a single GPU into up to seven separate GPU Instances for CUDA applications, providing multiple users with isolated GPU resources for optimal utilization. For example, the NVIDIA A100 GPU supports 19 possible configurations, offering flexibility in resource allocation based on workload requirements. Each MIG partition contains portions of Streaming Multiprocessors (SMs) and memory slices. Importantly, MIG partitioning extends beyond just compute and memory resources. L2 cache banks, memory controllers, and DRAM address buses are also part of the partitioning scheme, ensuring isolation of resources, which provides several benefits:

- **Error Isolation**: Unlike MPS, MIG does not use a shared GPU context. This separation ensures that errors in one partition do not affect applications running in other partitions, enhancing system reliability.
- **Performance Predictability**: Hard partitioning guarantees that each instance has access to its allocated resources, eliminating interference between multiple CUDA applications.
- **Security**: The strict isolation between partitions enhances security, making MIG suitable for multi-tenant environments where data privacy is crucial.

However this hard-partitioning also has downsides. Once the partitions are created, unused resources in one partition cannot be reclaimed by other partitions [74]. In addition repartitioning is not dynamic. It requires all applications to stop in order to apply the new configuration, making it less suitable for cases where the configurations need to be updated frequently.

It's worth mentioning that MPS and MIG are complementary technologies. While MIG provides hard partitioning at the hardware level, MPS can be used within individual MIG partitions to further optimize resource utilization. This combination allows for fine-grained control over GPU resources, enabling administrators to balance isolation requirements with efficient resource sharing.

2.4 Existing Inference Systems

The rise of ML inference in various applications has led to the development of numerous ML inference serving systems in recent years. We have reviewed some of these systems, evaluating the feasibility of utilizing them as the foundation for our research. Our assessment focused on the effort required to establish a functional setup that would facilitate the evaluation of different placement policies. For some of these systems, several implementation questions remained at the end of our investigation, that would each have been subject to further time consuming investigations. We concluded that developing a custom, minimally functional system would best serve our research objectives. This approach allowed us to focus on the core functionality required for our study, providing greater flexibility in implementing and testing various placement policies. Nevertheless, we provide an overview on two of the most popular ML inference serving systems below.

2.4.1 RayServe

RayServe [51] stands out as a widely adopted open-source model serving library built upon the Ray framework. Its framework-agnostic nature allows for seamless integration with various Python-based deep learning frameworks, including PyTorch, TensorFlow, and Keras. RayServe is engineered to scale across large heterogeneous clusters of machines and supports the colocation of applications on GPUs. Furthermore, it offers advanced features such as dynamic batching to optimize inference performance.

2.4.2 NVIDIA Triton Inference Server

The NVIDIA Triton Inference Server [31] has recently attracted increasing attention within the research community and has been utilized in notable works such as iGniter [81]. Similar to RayServe, the Triton Inference Server offers support for multiple machine learning frameworks, including PyTorch, TensorFlow, ONNX, and custom models. It addresses critical challenges in the field of ML inference, such as maximizing GPU utilization and delivering low-latency responses. The server also incorporates advanced features like dynamic batching to enhance inference efficiency and supports heterogenous clusters.

Chapter 3

Related Work

This chapter provides an overview on related work relevant for ML inference schedulers.

3.1 Overview of ML Inference Schedulers

Table 3.1 presents a comprehensive overview of the machine learning (ML) inference schedulers examined in this study. To denote various optimization objectives, we employ the following symbols: latency (♣), throughput (♠), utilization (♢), cost (♡), accuracy (∞), simplified deployment (△), and carbon emissions (□). The table delineates several key characteristics for each scheduler, including implementation of GPU sharing techniques, utilization of batching as a method to enhance utilization, awareness of potential interference, and support for multi-GPU configurations.

Scheduler	**Objectives**	**GPU Sharing**	**Batching**	**Profiling**	**Intf. Aware**	**multi GPU**
Clipper [61]	♣♠∞△		✓			✓
Space Time [68]	♠♢	spatial				
Ebird [62]	♣♠♢		✓			✓
Clockwork [66]	♣♠	temporal	✓	✓		
GSLICE [64]	♠♢	spatial	✓			
Irina [80]	♣♠♢	spatial	✓	✓		
Abacus [63]	♣♠♢	spatial		✓	✓	✓
INFaaS [75]	♣♠♡△		✓	✓	✓	
Interference-Aware Scheduling [72]	♣	spatial		✓	✓	
MIG-Serving [77]	♡	spatial	✓			✓
Morphling [79]	♠△		✓	✓		✓
Gpulet [60]	♣♠♢	spatial + temporal	✓	✓	✓	✓
AlpaServe [71]	♣♠	temporal		✓		✓
Clover [70]	♣□	spatial		✓		✓
iGniter [81]	♣♢♡	spatial	✓	✓	✓	✓
Shepherd [82]	♠♢	temporal	✓	✓		✓
Usher [76]	♣♠	spatial	✓	✓		✓

Table 3.1: Overview of ML Inference Schedulers on GPUs

3.2 Clipper: A Low-Latency Online Prediction Serving System

Clipper [61] addresses the challenge of improving the latency, throughput, and accuracy of machine learning serving. The authors highlight the complexity and reduced accuracy in deployment caused by the multitude of machine learning frameworks, each optimized for specific models or domains. They argue that this tight coupling of applications to these frameworks hinders the rapid and iterative development of new models and techniques. To overcome this, Clipper introduces an intermediate abstraction layer, consisting of two main components. Firstly, a model abstraction layer provides a common prediction interface, enabling the easy introduction of new models without modifying end-user applications. Each model is encapsulated within a separate Docker container, communicating with Clipper through an RPC mechanism for uniform interface and simplified deployment. Secondly, a model selection layer, built on top of the model abstraction layer, aims to enhance accuracy by dispatching inference queries to one or more models and aggregating the results for final predictions. Additionally, to increase throughput, the model abstraction layer employs dynamic adaptive batching to distribute inference requests across multiple model containers without violating latency Service Level Objective (SLO) requirements. This architecture streamlines the deployment process, decoupling applications from the variability and diversity of machine learning frameworks while improving overall performance and adaptability.

3.3 Dynamic Space-Time Scheduling for GPU Inference

Jain et al. [68] present a scheduler for ML prediction workloads that addresses the challenge of poorly utilized GPUs, using a combination of temporal and spatial sharing. Their idea is to merge many small kernels from disjoint DNN graphs into a small set of larger super kernels that together are capable of saturating all resources on the GPU for a given timeslice.

3.4 Ebird: Elastic batch for improving responsiveness and throughput of deep learning services

Ebird [62] introduces a scheduling mechanism designed to enhance the responsiveness and throughput of machine learning (ML) inference tasks conducted on GPUs. The authors identify issues with static batch sizes, commonly used to optimize GPU utilization, which result in prolonged latencies during low-load periods as incoming requests wait for batch completion.

Conversely, during high-load periods, GPUs often remain underutilized due to sequential batch processing, resulting in idle time during data transfer between CPU and GPU. To mitigate these challenges, Ebird advocates for elastic batching, a dynamic approach leveraging CUDA streams to concurrently process inference batches of varying sizes. Ebird maintains an in-memory pool on the GPU to swiftly store incoming inference data upon arrival, enabling overlapping of data transfer and computation. A multi-granularity inference engine then dispatches inference requests from this pool in diverse batch sizes to available workers (CUDA streams), ensuring efficient resource utilization and reduced latency across varying workload intensities.

3.5 Serving DNNs like Clockwork: Performance Predictability from the Bottom Up

Clockwork [66] highlights the prevalent issue of over-provisioning resources in DNN serving systems to face the high variability of requests under tight latency Sercive Level Objectives (SLO). The authors argue however that many of these DNN models are based on deterministic computations that make them very predictable. The variability in response times primarily stems from internal system components across different layers of the stack, exacerbated by best-effort techniques attempting to mitigate these variations. Clockwork's approach therefore involves eliminating sources of unpredictability within the system by centralizing configuration and scheduling decisions across all layers. Through a central controller, Clockwork actively manages memory allocation and de-allocation on worker nodes and restricts concurrent execution to mitigate interference. While Clockwork successfully meets most latency requirements, certain design choices lead to a trade-off with hardware utilization and throughput.

3.6 GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform

GSLICE [64] addresses the underutilization of modern GPUs by deep neural network (DNN) workloads. The authors highlight that allocating additional resources beyond a certain threshold, termed the *kneepoint*, yields minimal benefits. To enhance GPU utilization, they propose spatially sharing GPUs across multiple parallel workloads. While solutions like NVIDIA MPS allow resource sharing, they lack adaptability to dynamically repartition resources based on workload fluctuations without substantial downtimes. GSLICE introduces a framework that extends NVIDIA MPS capabilities, enabling dynamic resource partitioning. It achieves this by reallocating GPU resources in the background to *stand-by inference functions*, ensuring rapid activation with minimal downtime. Additionally, GSLICE implements an adaptive batching mechanism to enhance throughput and GPU utilization while maintaining latencies below Service Level Objectives (SLOs). To minimize data transfer time to GPU memory, GSLICE leverages GPU Direct Memory Access (DMA) to directly scatter-gather data from network packets. Furthermore, it adopts a parameter sharing strategy to reduce memory footprint and facilitate serving multiple models on a single GPU.

3.7 Irina: Accelerating DNN Inference with efficient Online Scheduling

Most schedulers that aim to improve the throughput or hardware utilization of inference requests assume predictable workload and present deteriorating performance as the workload becomes unpredictable. Wu et al. [80] present Irina, an inference scheduler whose primary goal is to minimize the average job completion time under unpredictable workload. In order to achieve this, Irina uses three mechanisms. *opportunistic batching* groups inference request to be handled by the same model into batches in order to utilize available compute resources in the best possible way and reduce inference delay. *Online job stacking* is used to collocate short and long running workloads onto the same GPU given that their peak memory consumption does not exceed the available amount of memory and latency SLO are not violated. In this decision process Irina however doesn't take into account the mutual interference that both models may create towards each other. For both methods, offline model profiling is conducted in order to gather the necessary information. Lastly, since preemption is known to improve job completion time and query response time, the framework uses *dynamic job preemption* by modifying the underlying model computation graph. It introduces so-called *exit* nodes between any two consecutive layers such that the requests can be actively preempted after any operator. The *exit nodes* contain information about the dynamic GPU memory allocation allowing it to safely free it upon preemption. The downside however is that the framework discards all intermediate computations up to the point of preemption. The authors claim it is sometimes easier to simply redo all of the computations rather than swapping intermediate results between the host and device.

3.8 Abacus: Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction

Abacus, as detailed in the paper by Cui et al. [63], aims to enhance the efficiency of DNN inference by improving throughput and latency. The primary challenge identified by the authors is the unpredictable and unstable latency associated with these requests, which complicates the effective collocation of different DNN workloads on GPUs without violating their Quality of Service (QoS) requirements. This unpredictability stems from the sensitivity of requests to their input data and the random overlap of different operators/kernels due to their varying arrival times. To address this challenge, Abacus introduces the concept of so-called *operator groups* to overlap operators and kernels from multiple user requests on a GPU within a scheduling round. By leveraging offline profiling, the framework trains a multi-layer perceptron capable of accurately predicting the latency of a given operator group. During each scheduling round, Abacus selects the user request with the least QoS headroom and adds its remaining operators to the group. It continues this process, adding operators from other user requests, as long as the QoS requirements can still be met. The authors assert that due to the asynchronous execution of GPU operations alongside host operations, the planning of an operator group for the next round can be seamlessly hidden by the execution of the operator group in the previous round, incurring no additional overhead.

3.9 INFaaS: Automated Model-less Inference Serving

The INFaaS framework [75] offers a solution to streamline the deployment of machine learning (ML) models, optimize costs, and enhance resource utilization. Recognizing the complexities involved in deploying ML applications and the challenge of determining the optimal configuration for serving inference requests, INFaaS introduces a model-less interface. This interface allows users to register a model without the need for pre-defined configurations. INFaaS automates the process of creating multiple variants of the registered model and profiles them by adjusting parameters such as batch size and underlying hardware. When submitting requests, users only need to specify constraints, and INFaaS leverages an integer linear programming model to optimize and select the most suitable variant for serving each request. Furthermore, the framework dynamically scales up resources as demand increases, ensuring efficient handling of peak loads. During periods of low demand, resources are shared across models to enhance resource utilization.

3.10 Interference-Aware Scheduling for Inference Serving

Mendoza et al. [72] emphasize that ML inference latency is highly impacted by the collocation factor and the hardware executing the model. Profiling each model-hardware combination for accurate latency predictions is impractical due to computational complexity. Instead of solely profiling various collocation configurations across hardware types and fitting a model, the authors propose learning a mapping between inference model characteristics (e.g., CPU/GPU utilization, memory footprint, number of input/output layers etc.), machine types, and resulting collocation latency degradation. By incorporating machine types and model characteristics into the input, they aim to exploit similarities in different collocation configurations across models and hardware to enhance prediction accuracy. While the authors demonstrate latency degradation improvements over least-load schedulers, the paper lacks discussion on the scheduler's impact on throughput and resource utilization.

3.11 Serving DNN Models with Multi-Instance GPUs: A case of the Reconfigurable Machine Scheduling Problem

MIG-SERVING [77] explores the utilization of Multi-Instance GPU (MIG) hardware support provided by NVIDIA to enhance the efficiency of A100 GPUs in serving deep neural network (DNN) inference requests, thereby improving cost-effectiveness. It aims to optimize GPU partitioning into smaller heterogeneous hardware instances and the scheduling of DNN services onto these partitions. This optimization is conducted considering constraints such as required Service Level Objectives (SLOs) and throughput rates. The framework allows for partial reconfiguration of GPU instances and applies repartitioning seamlessly in response to changing requirements. The optimization algorithm powering *MIG-SERVING* takes minutes to hours, depending on the desired level of optimality, making frequent model or SLO updates impractical. While the paper reports savings of up to 40% in the number of GPUs required for certain workloads, it doesn't consider the difference in workspan latency among approaches. The baseline achieves faster completion times within seconds, whereas various versions of MIG-SERVING, aimed at more cost-efficient deployments, operate on the order of minutes to hours due to the intensive optimization process. Furthermore, according to the parper, the time to re-partition is in the order of minutes. Underutilized resources in periods of low load cannot be temporarily reclaimed and made available to services experiencing high load, leading to a waste of resources. The framework, therefore, might prove less beneficial in the presence of bursty load.

3.12 Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving

Morphling [79] delves into the significant impact of system configuration, such as GPU type, GPU memory, CPU cores, and hyperparameters like batch size, on the throughput of many ML inference services. Unlike existing deployment frameworks such as Clipper [61] or INFaaS [75], which primarily focus on abstracting model heterogeneity into a common interface and selecting the appropriate model variant for serving user requests, Morphling takes a different approach. It introduces a framework that formulates the system configuration tuning process as an optimization problem. Initially, a base model is trained offline to capture general performance configuration trends across a range of ML models. When registering a new inference service, this base model serves as an initial regression model. Through online fine-tuning using few-shot regression, Morphling dynamically adjusts and refines the model to explore the configuration space and quickly converge to near-optimal configurations.

3.13 GPUlet: Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing

Choi et al. [60] propose novel strategies to address the limitations of traditional batching and temporal sharing techniques when utilizing GPUs for ML inference services. They contend that unlike in training scenarios, where data is readily available, inference data arrives at varying rates and times, challenging efficient GPU utilization. Specifically, under strict Service Level Objectives (SLOs), schedulers face a narrow window to aggregate requests into batches, often resulting in suboptimal batch sizes that fail to fully leverage GPU compute resources. To overcome these challenges, the authors advocate for exploiting hardware capabilities to spatially partition GPUs into smaller virtual units called *gpulets*, offering finer-grained scheduling. Leveraging offline profiled data and continuous monitoring of input arrival rates, their framework dynamically assigns ML services to different gpulets. Within each gpulet, temporal sharing is applied to overlap the buffering of incoming requests from multiple services into batches and serving them during subsequent time slots. Moreover, the scheduler continuously monitors arrival rates across services, allowing it to not only adjust the number of active GPUs required to meet all requests optimally but also to dynamically repartition GPUs into different configurations.

3.14 AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving

AlpaServe [71] employs model parallelism in ML serving systems to enhance inference throughput and reduce latency, even for models that could theoretically fit on a single GPU. Despite the overhead associated with model parallelism compared to exclusive GPU placement, the authors demonstrate its efficacy in improving model serving, particularly during periods of low and bursty load, limited device memory, or tight Service Level Objectives (SLOs). In AlpaServe, for a given cluster of GPUs and a given set of models, the models are partitioned into smaller stages, with stages from different models collocated on the same GPU. Compared to the exclusive placement where requests are processed on a single GPU, AlpaServe adopts a pipelined approach where requests are served by all GPUs of the cluster. Each cluster maintains a queue of incoming requests, which are dispatched on a first-come, first-served basis. During periods of high load for a particular model, while others experience lower demand, the popular model can get a higher share on the computational resources through statistical multiplexing of the devices. However, it's important to note that the advantages of statistically multiplexing diminish as the load on all models increases or SLOs become looser. In such scenarios, the overhead introduced by model parallelism begins to outweigh its benefits, and exclusive placement of models on GPUs resutls in better performance.

3.15 Clover: Toward Sustainable AI with Carbon-Aware Machine Learning Inference

ML inference services currently represent a significant portion of cloud-hosted applications and contribute substantially to the carbon footprint of datacenters due to their high energy consumption. Addressing this issue, Clover [70] focuses on reducing the carbon footprint of inference services by intelligently creating a mixture of model variants on selected GPU partitions to optimize for both carbon footprint and inference accuracy under Service Level Objective (SLO) constraints. Clover leverages NVIDIA MIG to partition GPUs into smaller instances, enhancing resource utilization. Additionally, it compromises a certain level of model accuracy to achieve better performance and significant reductions in energy consumption. However, rather than treating these approaches as separate dimensions, Clover approaches them as a unified optimization problem. It incorporates lower-quality models to mitigate the interference introduced by spatially co-locating models on the same GPU. This integrated approach enables Clover to reduce the carbon footprint of ML inference services while ensuring SLA requirements are met, with only minimal sacrifices in accuracy.

3.16 iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud

iGniter [81] highlights the shortcomings of existing frameworks designed to optimize GPU resource utilization using spatial co-location of inference workloads without consideration for interference. The authors identify three primary sources of interference that lead to performance degradation. Firstly, the GPU scheduler introduces additional delays by scheduling kernels from various workloads in a round-robin manner, with scheduling overhead increasing as more workloads are co-located and kernels per workload increase. Secondly, despite the ability of NVIDIA MPS to distribute streaming multiprocessors among co-located workloads, contention in shared resources such as the L2 cache are the main drivers of the active kernel execution time. Lastly, performance deterioration occurs due to frequency reduction as the GPU adjusts its frequency to manage increased workload demands and maintain power limits. In response to these challenges, iGniter introduces a lightweight analytical performance predictor that considers all three interference sources to accurately predict the latencies of different workloads under co-location scenarios. Leveraging this predictor, iGniter formulates a cost-effective GPU resource provisioning strategy by heuristically assigning workloads to GPUs that minimize interference overhead while ensuring to not violate Service Level Objectives (SLOs). Notably, iGniter's approach is agnostic to GPU architecture, making it adaptable to clusters of heterogeneous GPUs.

3.17 Sheperd: Serving DNNs in the Wild

Zhang et al. [82] delve into the challenges of resource provisioning in the face of unpredictable and bursty request streams. They highlight the difficulty in effectively allocating resources due to the unpredictable request arrivals in the short term. This often results in over-provisioning and underutilization of resources. In response to this challenge, the authors introduce Shepherd, a system designed to achieve high goodput, even in the presence of unpredictable workloads, while ensuring scalability and efficient resource utilization. Shepherd adopts a two-level scheduling framework to address these goals. Firstly, a periodic planner groups individual request streams into moderately-sized clusters, typically containing hundreds to thousands of streams. This grouping enhances predictability, making resource provisioning more manageable at the cluster-level scale. Secondly, an online serving scheduler schedules requests across streams within each serving group independently. It temporarily shares the GPUs within each serving group and favors to execute large batch sizes over smaller ones in order to maximize goodput. To correct for sub-optimal scheduling decisions made in the past, the online scheduler uses software-level preemption to preempt a current running batch in favor of a larger queued batch if their sizes differ significantly. The authors prove that it is not possible to provide goodput guarantees without using preemption when not knowing the future traffic arrival patterns in advance. The 2-level scheduling approach adopted by Sheperd has two advantages. It reduces the decision space that the online serving phase has to operate in while at the same time still provisioning an adequate number of streams and GPU workers to maximize utilization.

3.18 USHER: Holistic Interference Avoidance for Resource Optimized ML Inference

Shubha et al. [76] present USHER, an end-to-end inference serving system that optimizes GPU computation and memory utilization through spatial multiplexing of resources in an interference-aware manner. USHER operates in three stages. In the first stage, the framework estimates the resource and memory requirements of a model. Instead of running the model on the GPU and profiling it explicitly, the authors train regression models that leverage the repetitive nature of a small set of operators in both convolution and transformer-based models to estimate compute and memory requirements at a kernel-level granularity. This approach allows them to classify models as either compute-heavy, memory-heavy, or neither. In the second stage, USHER uses these estimations to group models into moderate-sized clusters, maximizing the likelihood of spatially co-locating compute-heavy and memory-heavy models within a group. The authors found that within such a group, replicating a model multiple times with smaller batch sizes across multiple GPUs, even when a single GPU could handle the workload within the latency SLO, can lead to a reduction in GPU fragmentation and thereby an increase in GPU utilization. In the third stage, USHER aims to minimize cache interference between co-located models. It does this by leveraging the fact that models of the same architecture often share operators with similar weight matrices. USHER merges the computation graphs of multiple deep learning models so that matrix multiplications of different models are performed simultaneously when a shared or similar weight matrix is present in the cache. The authors report that this merging process results in negligible loss of accuracy.

Chapter 4

Architecture

In this chapter, we will introduce our cluster scheduler infrastructure. Section 4.1 provides a high-level overview of the architecture and its core concepts. Section 4.2 provides the definition of classes that will be referenced all over the chapter. Section 4.3 details the components operating at the cluster level, while section 4.4 focuses on the node level of the cluster. Section 4.5 will delve into the worker level of the cluster. We will discuss the networking setup of the cluster in section 4.6, and conclude this chapter with an overview of its current limitations in section 4.7.

4.1 Overview

Recent cluster schedulers targeting running ML inference on GPUs have frequently introduced their own architectures, often comprising multiple tightly coupled components. This tight integration makes it difficult to modify or replace individual components, such as the placement policy, without affecting the entire system. Consequently, extracting the placement policy from one system and incorporating it into the architecture of another system is non-trivial. As a result the performance comparison between systems is tricky.

To address this issue, we propose a system that abstracts the core components of schedulers for ML inference, offering a modular approach that allows for the seamless integration of various placement policies. This challenge of inflexible architectures has also been recognized in the context of cluster schedulers for deep learning training. For instance, the authors of Blox [58] developed a modular toolkit to enable more flexible construction and evaluation of deep learning training schedulers. Similarly, our system aims to provide a more adaptable framework for ML inference, facilitating easier experimentation and performance comparison across different placement strategies.

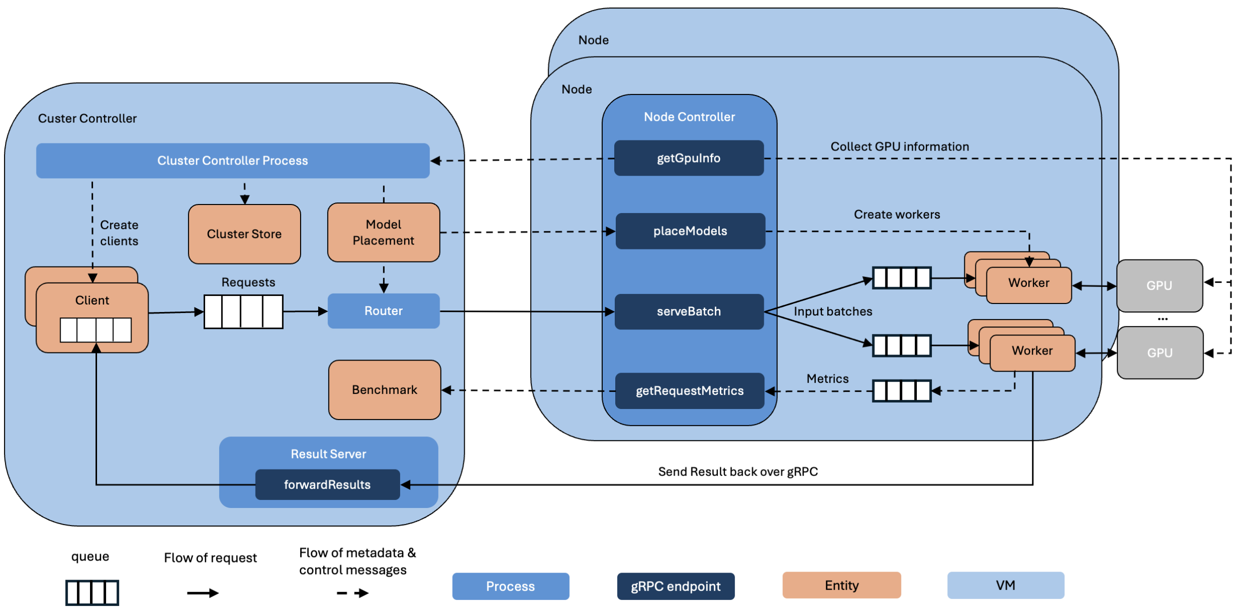


Figure 4.1: Architecture Overview

The resulting architecture can be seen in figure 4.1. There are 3 main levels in the system architecture:

1. **Cluster Level**: At the very top level there is a cluster. A cluster always consists of one `Cluster Controller` and at least one node. The `Cluster Controller` is an entity that is responsible for coordinating the work across all nodes and exposing an interface for clients to submit requests to.

2. **Node Level**: At the mid level there are nodes. There is one node per machine, and a single node always consists of one `Node Controller` and at least one GPU attached to it. Each GPU can have multiple workers assigned to it. The job of the `Node Controller` is to coordinate all workers and expose a communication interface to the `Cluster Controller`.

3. **Worker Level**: At the bottom level there are workers. A `Worker` is always assigned to one GPU. It is responsible for launching the inference requests on the GPU. Multiple workers can be attached to the same GPU and they live on the same machine as their managing `Node Controller`.

The architecture is build up in a way that it can run both on a single machine or distributed across multiple machines, on premise or in the cloud. There are however two restrictions, that we will list below:

1. All functionality related to the `Cluster Controller` has to run on the same machine as its component will use shared memory to communicate with each other. We will share more details in section 4.3.
2. All functionality related to the `Node Controller` has to run on the same machine as again its components will rely on shared memory to communicate with each other.

With the above constraints introduced, the following are all valid constellations.

- **Single Machine Scenario**: We have a cluster that consists of one node with $n \geq 1$ GPUs attached to it. There is one `Cluster Controller` and one `Node Controller` that will both live on the same machine.
- **Multi Machine Scenario**: We have a cluster that consists of one machine running the `Cluster Controller` and $m \geq 1$ additional machines each running a node. Each node i can again have $n_i \geq 1$ GPUs attached to it.

The whole system is implemented in Python using PyTorch as backend for running the inference and the implementation is available on `https://github.com/eth-easl/orion_dev`.

We will next spend an individual section for each of the three levels and explain its components in more detail. Until now we have highlighted the main entities `Cluster Controller`, `Node Controller` and `Worker`. From now on we will no longer explicitly highlight them for readability reasons, but will always refer to them as cluster controller, node controller or worker respectively.

4.2 Class Definitions

In this section, we introduce several key classes that will be referenced repeatedly in the subsequent sections. The class definitions in Listing 4.1 provide simplified representations of their actual implementations but are sufficient to understand the general functionality of the system. Further details about these classes will not be provided at this point. Instead, they will become clearer in their respective contexts throughout the document.

4.3 Cluster Level

This section is dedicated to explain the main components of the cluster level. We will again start by first providing an overview on the high level task of each component, how they interact with each other and then focus

```python
class Model:
    model_name:   str
    model_type:   str
    target_rps:   int
    slo:          int

class Client:
    model:        Model

class Node:
    uid:          str
    ip:           str
    port:         str

class GPU:
    uid:          str
    local_id:     int
    gpu_type:     str
    memory:       int
    sm_count:     int
    node:         Node

class ModelPlacement:
    model:        Model
    gpu:          GPU
    batch_size:   int
    mps_share:    Optional[float]
```

Listing 4.1: Domain Model of our ML inference cluster

more closely on each of them individually. Figure 4.1 provides the necessary overview.

- **Cluster Controller Process**: The main orchestrator of the cluster, responsible for coordinating all components as outlined in Listing 4.2. This process operates independently, ensuring all subsystems function cohesively. Further details are provided in Section 4.3.1
- **Model Placement Policy**: Integrated within the cluster controller process, this component executes the model placement policy. Its interface is designed to be modular, allowing easy swapping of placement algorithms. Section 4.3.2 delves deeper into its implementation.
- **Router**: The router handles the routing and load-balancing of client-submitted inference requests, directing them to the appropriate nodes for processing. A single router instance runs independently as its own process, as described in Section 4.3.3.
- **Result Server**: Acting as a proxy, the result server forwards inference results from the workers back to the clients. It operates as a standalone

process, ensuring that communication between workers and clients is streamlined.

- **Cluster Store**: A shared memory-based database that holds the state of the entire cluster. This allows the cluster controller process, router, and result server to share and access overall information efficiently.
- **Client**: These entities generate load for the cluster by submitting requests. They operate as independent processes on the same machine as the other cluster components. A detailed discussion of their current state and potential improvements is provided in Section 4.3.6.
- **Benchmark**: This component runs at the final stage, just before resource cleanup, collecting and aggregating performance data across the cluster. It provides key metrics for evaluation and debugging and operates as part of the cluster controller process.

4.3.1 Cluster Controller Process

The cluster controller process serves as the core and entry point of the cluster. It is responsible for launching all cluster-level components, ensuring their coordination and communication, and terminating them upon completion. Listing 4.2 outlines the primary sequence of actions undertaken by this process.

The algorithm takes as input an array of `clients` and an array of `nodes`. We refer the reader back to section 4.2 for the definition of the `Client` and `Node` class.

The algorithm begins by initializing the cluster store in line 5, which holds shared information accessible to all cluster-level processes. In line 6 the cluster controller gathers information on all GPUs in the cluster. Using gRPC, it contacts each node to retrieve information on the GPUs attached to it, including the GPU's local identifier, type, memory capacity, and the number of Streaming Multiprocessors (SMs). This data is stored in the cluster store. It is important to note that the nodes must be operational by the time the cluster controller contacts them.

In lines 8 to 11, the algorithm proceeds by creating a shared client submission queue for all clients to submit requests to, and a client result queue for each client to receive the results of its requests.

Subsequently, the cluster controller invokes the model placement policy in line 13, providing it with information on the clients to be served. The policy returns a mapping of models to the GPUs that will host them, along with corresponding batch sizes. The internal logic of the placement policy is abstracted from the caller, enabling the easy replacement of any policy that adheres to the interface. After receiving the model placement mapping, the cluster controller shares the relevant information with each node over gRPC

```
 1 def run_cluster_controller(
 2     clients: list[Client],
 3     nodes: list[Node]
 4 ):
 5     cluster_state = init_cluster_state()
 6     gpus = fetch_gpus_from_nodes(nodes)
 7 
 8     client_sub_queue = shared_queue()
 9     client_res_queues = map()
10     for c in clients:
11         client_res_queues[c] = shared_queue()
12 
13     model_placement = run_placement_policy(clients, gpus)
14     send_placement_to_all_nodes(nodes, model_placement)
15 
16     start_router(client_sub_queue, model_placement, nodes)
17     start_result_server(client_res_queues)
18 
19     for c in clients:
20         start_client_sender(c, client_sub_queue)
21         start_client_receiver(c, client_res_queues[c])
22 
23     # wait for client sender and receivers to finsih
24     client_senders.join()
25     client_receivers.join()
26 
27     # collect metrics an clean up
28     run_benchmark()
29     clean_up_all_resources()
```

Listing 4.2: Cluster Controller Process

in line 14. The nodes operate independently and have no awareness of each other's presence.

In lines 16 and 17 the cluster controller then starts the result server and the router, granting them access to the client result queues and client submission queue respectively.

Once the cluster is fully set up and running, the client processes are launched in lines 19 to 21, each as an individual sender and receiver process, and the serving of requests will start. After all requests have been served, the benchmark module is launched to collect metrics across the whole cluster in line 28 and finally all resources are cleaned up.

4.3.2 Model Placement Policy

This subsection will provide more details on the model placement policy module and its abstraction. The interface that each policy needs to implement is provided below:

```
run_placement_policy(clients: list[Client], gpus: list[GPU])
    -> list[ModelPlacement]
```

We refer the reader back to section 4.2 for the corresponding class definitions of `Client`, `GPU`, and `ModelPlacement`. For each client, the most important attributes are its `model_name`, the target request submission rate `target_rps` (in requests per second), and its Service Level Objective `slo`, specified in milliseconds.

A GPU consists of a universal `uid` and a `local_id` corresponding to its local identifier on its respective machine. The `node` refers to the node object that the GPU is attached to. As a consequence, all GPUs on the same machine share the same `node.uid`. A GPU can thus be uniquely identified either by its global `uid` or by the tuple (`node.uid`, `local_id`). This abstraction gives the placement policy full visibility into the cluster's topology. While the placement policies evaluated in this work assume models that can be served by a single GPU, future work may explore scenarios where larger models are split across multiple GPUs. In such cases, it is preferable to allocate models across GPUs within the same node. This abstraction should not require any changes to also support these cases.

The placement policy is required to return a list of `ModelPlacement` objects. A model placement specifies that a particular model will be served on a specific `gpu`, with a designated `batch_size` and an optional `mps_share`. An `mps_share` of 50% for example indicates that the model should only have access to 50% of all the Streaming Multiprocessors (SMs) on the corresponding GPU. It is important to note that a model may be served on multiple GPUs, with different batch sizes and resource allocations. In such cases, the policy returns a separate `ModelPlacement` object for each GPU. From this point forward, we refer to these individual placements as **model replicas**.

We will not focus on any specific implementation of a placement policy here, but will delay this to chapter 5.

4.3.3 Router

We are next going to focus on the router in more detail, a crucial component of the overall scheduler system. The router's task consist in distributing the clients' inference requests among all the corresponding model replicas. To achieve this, the router is provided with the client submission queue *client_sub_queue* from which it can dequeue the client requests, the model placement that contains a mapping from each model to the GPUs serving it and the addresses of all the nodes in the cluster. The functionality of the router is provided by listing 4.3.

The router is responsible for grouping the individual requests into batches before sending them to the nodes. The desired batch size for each model can differ between each of the replicas and is provided by the `model_placements`.

```
 1 def run_router(
 2     client_sub_queue: shared_queue,
 3     model_placement: list[ModelPlacement],
 4     nodes: list[Node]
 5 ) -> None:
 6     cur_replica = get_first_replica_per_model(model_placement)
 7     batch_per_model = map()
 8     batch_timeout = map()
 9
10     while not interrupted:
11         req = get_next_request(client_sub_queue)
12
13         if req.model in batch_per_model:
14             batch_per_model[req.model].add(req)
15         else:
16             batch_per_model[req.model] = [req]
17             batch_timeout[req.model] = now() + MAX_WAIT_TIME
18
19         if batch_is_full() or timeout_reached():
20             send_batch_to_node(cur_replica[req.model].node)
21             batch_per_model.pop(req.model)
22             cur_replica[req.model] =
                   round_robin_to_next_replica()
23
24     send_remaining_open_batches_to_nodes()
```

Listing 4.3: Router

The router will round-robin the batches for each model across its replicas. To do so, it maintains three important maps:

1. *cur_replica*: For each model the router is always building at most one batch at a time designated to a specific replica. The replica under current consideration is stored by the router in the *cur_replica* map.

2. *batch_per_model*: This map stores a list of the requests part of each batch that is being constructed.

3. *batch_timeout*: To prevent that requests are being held up at the router for too long while waiting to reach their target batch size, the router maintains a timeout for each batch after which it will at the latest be send off to its corresponding worker. A similar behavior is adopted by works like Ebird [62], Clockwork [66], Irina [80], iGniter [81] and the Triton Inference sever [55].

The router continuously polls the `client_sub_queue` for incoming requests. For each request, the router, in line 13, checks whether a batch is currently being constructed. If a batch is in progress, the request is added to it. If not, the router initializes a new batch and sets a timeout. This timeout is determined by the user-configurable constant `MAX_WAIT_TIME`, which we have set to 100ms. This value, aligns to our understanding with the default configuration used in iGniter [55, 81]. Even though this constant may seem marginal, we will show in chapter 6 that its value can have a significant impact on the overall performance of the system.

After adding the request to the batch, the router will evaluate whether the current batch has reached its desired size or if its timeout has expired in line 19. If either condition is met, the batch is sent to the node hosting the corresponding model replica using gRPC. For more details on the networking setup, we refer the reader to Section 4.6. Once the batch is sent, it is marked as resolved in line 21, and the router proceeds to the next model replica in a round-robin manner in line 22. This ensures that each replica handles a portion of the total load proportional to its designated batch size.

In the event of an interruption or termination, the router, in line 24, sends any remaining batches, regardless of whether they have reached the desired size or their timeouts have been triggered.

Synchronous vs Asynchronous Router

The `send_batch_to_node` function is responsible for transmitting a list of requests over the network to the node. The overhead of network transmission primarily depends on the amount of data being sent and the available bandwidth to the router. If this transmission were handled synchronously, the router would be blocked for the entire duration of the transmission, preventing it from processing requests in the `client_sub_queue`. This could significantly reduce the router's throughput. To mitigate this issue, we implemented the router asynchronously using asynchronous gRPC [39], aiming to hide the network overhead and preventing the router from becoming a bottleneck in the cluster.

The router operates as a single Python process with multiple Python coroutines [37]. When a coroutine asynchronously invokes the `send_batch_to_node` function, it is blocked during the network request. However, instead of halting the entire router process, the coroutine yields control back to the event loop. This allows the event loop to schedule another coroutine in the meantime and continue constructing batches while the network transmission proceeds in the background.

The number of coroutines can be set by the user and should be carefully selected based on the number of model replicas being served. As a general guideline, the number of coroutines should be at least equal to the number of model replicas. If the number of coroutines is fewer than the number of replicas, all coroutines may become blocked while waiting for their network transmissions to complete, causing the router to stall until the first coroutine finishes, delaying further batch construction.

4.3.4 Result Server

The result server is a gRPC server that is a proxy for forwarding the inference results from the workers back to the clients. The workers at the node level have no knowledge on how to reach the clients. On the contrary the result server is given a list of client result queues *client_res_queues*. For each client it uses a dedicated queue to forward its results. Our system currently has one result server running for the entire cluster, but it may easily be replicated multiple times.

4.3.5 Cluster Store

The Cluster Store functions as a shared memory-based database, accessible to all components at the cluster level. Its primary role is to store information about the overall state of the cluster, such as GPU topology, model placements, and client data. It is implemented using Python Managers [40]. Due to the inherent performance overhead associated with accessing shared memory via a Python Manager, the store is designed to hold data that remains constant over time and accessed infrequently. It is not intended as a communication channel between processes, but rather as a repository for more static, less frequently modified data.

4.3.6 Clients

We will next focus on the client setup as it stands at the time of writing. Several simplifications have been made during the course of this work due to time constraints. While this setup is functional, it is far from ideal and will require careful redesign in the future. For each simplification, we will provide background context and propose improvements for future iterations.

Clients are the entities responsible for submitting load to the cluster. A client is characterized by three key properties:

- a **Model** for which the client submits requests,
- a **Submission Rate** that defines the target number of requests the client submits per second, and
- a **Service Level Objective (SLO)** that specifies the upper bound on the end-to-end (e2e) latency a request may take to be considered valid.

A client is implemented as two separate processes running in parallel. A sender process handles submitting requests to the cluster, while a receiver process waits for the results of these requests to be returned. We have decided to implement each task as a separate process to avoid that they interfere with each other.

Client Sender

```
def run_client(
    client_id: int,
    client_sub_queue: shared_queue,
    model: Model,
    num_requests: int,
    rps: int
) -> None:
    # make random generator deterministic across runs
    set_seed()

    for i = 1 to num_requests:
        req_id = generate_unique_id()
        input_shape = get_input_shape(model) # simplification
        client_sub_queue.put(
            req_id,
            input_shape,
            client_id,
            model
        )

        sleep_time = random_sample_from_exp(1 / rps)
        sleep(sleep_time)
```

Listing 4.4: Client Sender

The functionality of the client sender is illustrated in Listing 4.4. The client is instructed to submit a total number of requests `num_requests`, at a target submission rate `rps`. For each request, a unique identifier is generated, allowing the request to be tracked and identified at any point in the cluster. The sender then queries the input shape required for the given model. For instance, when performing object recognition with the `resnet50` model [50], the input shape would be $3 \times 224 \times 224$. It is important to note that the client only submits the input shape to the queue when sending the request to the cluster. The actual input data is generated at the worker node.

This is a key simplification made in this work, motivated by bottlenecks that we have observed in gRPC when transmitting large message sizes. We will provide further details on this topic in Section 4.6. After submitting the request to the queue, the sender sleeps for a duration sampled from an exponential distribution [8]. Note that the client sender does not wait for the result of its request to return before sending the next request, thus operating in an open system configuration.

It is important to distinguish between the **target request submission rate** (the number of requests per second the client is supposed to submit) and the **achieved request submission rate** (the number of requests per second the client actually submits). Ideally, the achieved request rate would match the target rate. However, in practice, the achieved rate is lower due to additional work being performed during each iteration, beyond just sleeping.

Table 4.1 compares the average achieved request submission rate measured over a 10-second period with the target submission rate. This experiment used four clients, all sharing the same `req_sub_queue`, and one router process. The measurements were conducted on a `n1-standard-16` machine [10] on the Google Cloud Platform, utilizing the `Intel Skylake` CPU platform. The table shows that in all 4 cases, the achieved submission rate is lower than the target submission rate. In order to increase the achieved submission rate, we can simply increase the target submission rate. While the achieved request submission rate will be our primary focus, we will always clarify which metric is being referenced.

	Target RPS			
Client ID	400	800	1200	1600
0	368	711	1022	1294
1	368	713	1022	1290
2	368	713	1022	1296
3	368	713	1022	1296

Table 4.1: Target request submission rate in request per second vs average achieved request submission rate (rounded to nearest integer) for 4 clients and one router over 10 seconds. Measurements have been conducted on a `n1-standard-16` machine with the `Intel Skylake` CPU platform in Google Cloud.

Client Receiver

The functionality of the client receiver is very simple. It keeps polling its dedicated `client_res_queue` and saves the timestamp of the arrivals.

Clients running on same machine as cluster-level components

In our current setup, clients submit their requests and receive corresponding results using shared queues. This configuration is possible because the clients, the router, and the result server are all running on the same machine, allowing them to share the queues. However, in a realistic deployment, clients would run on separate machines outside the cluster, communicating with the cluster over the network. The simplified setup was chosen to quickly develop a functional system that could be used to evaluate the impact of different placement policies on overall system performance. Due to time constraints, we were unable to decouple the clients from the cluster and run them as separate entities on different machines.

This simplification should be addressed in future work. For request submission, the solution is relatively straightforward. The router could become the entry point to the cluster by hosting a server that exposes an endpoint for clients to submit requests. Assuming this endpoint is accessible from outside the cluster, the client sender processes could easily run on a separate machine. However, handling the client receiver presents two potential solutions, each with its own advantages and disadvantages:

1. **Using the Result Server as a Proxy**: One option is to retain the result server and use it as a proxy to forward results from the workers to the clients. The advantage of this approach is that the result server remains within the cluster, maintaining access to the cluster store, which contains relevant information about the clients. This enables the result server to know how to contact individual clients. Additionally, the result server could extract and store performance data at the cluster level, such as request latencies, to monitor how long a request spends at the node or how long the actual inference takes. This feedback data could be valuable for the router when deciding where to send future batches. For instance, if the requests for a model on a particular worker take significantly longer than expected, this could indicate interference or an overloaded worker. The router could then dynamically load balance future requests to other workers until the situation stabilizes.

   This feedback could also inform the placement policy. Currently, the placement policy is executed only once at the beginning and remains static. Future work could explore a dynamic placement policy that runs periodically and incorporates feedback from the workers. For example, more replicas could be assigned to a model that struggles to keep up with demand, or replicas could be removed from models that are over-allocated. Systems like GSLICE [64] and iGniter [81] already use feedback data to make adjustments to model placement. The primary disadvantage of this solution is that it introduces another network hop in the end-to-end lifecycle of a request, adding additional network overhead. Another disadvantage is that the result server becomes a single point of failure. In the case of a failure at the cluster-level the clients have no way of retrieving their results.

2. **Direct communication between workers and clients**: Another approach is to have the workers send the results directly to the clients. As described in more detail in Section 4.5, each model replica is managed by a dedicated worker. This worker could be extended to include information about the clients, enabling direct communication with them. This method would likely provide the fastest way to return results from workers to clients, and since each worker is responsible for

its own results, there would be less risk of the result server becoming a bottleneck. However, this approach has some drawbacks. First, it would eliminate the valuable feedback data that could otherwise be shared with cluster-level components, as described earlier. Second, it introduces the concern of sharing more client-related information with workers than might be ideal.

4.3.7 Benchmark

The Benchmark component is not essential for the core functionality of the system but serves as a performance analysis tool. It runs as part of the cluster controller process before termination, after all client sender and receiver processes have terminated. As described earlier in subsection 4.3.6, client senders generate a unique identifier for each request they submit. This allows all components across the cluster that participate in serving the request to log timestamps for each stage of the process. After all requests have been processed, the benchmark component's role is to gather the timestamps recorded for each request by all the components in the cluster.

For components operating at the cluster level, a shared queue is used to send the timestamps to the benchmark. To collect the timestamps from the node and worker levels, the benchmark component communicates with each node controller over the network. The node controller is responsible for gathering the timestamps from all its workers before sending the data back to the benchmark. Using the unique request ids, the benchmark can merge the timestamps with relevant metadata, creating a comprehensive timeline for each request. For example metadata might include the batch id that a particular request was part of.

Once all timestamps are collected and merged, the benchmark provides a detailed performance overview of the cluster. This overview can be generated at varying levels of granularity, offering insights into the system's behavior and helping identify any performance bottlenecks or inefficiencies across the cluster.

4.4 Node Level

In this section, we will focus on the role of nodes within the overall cluster topology and break down their key components.

4.4.1 Node Controller

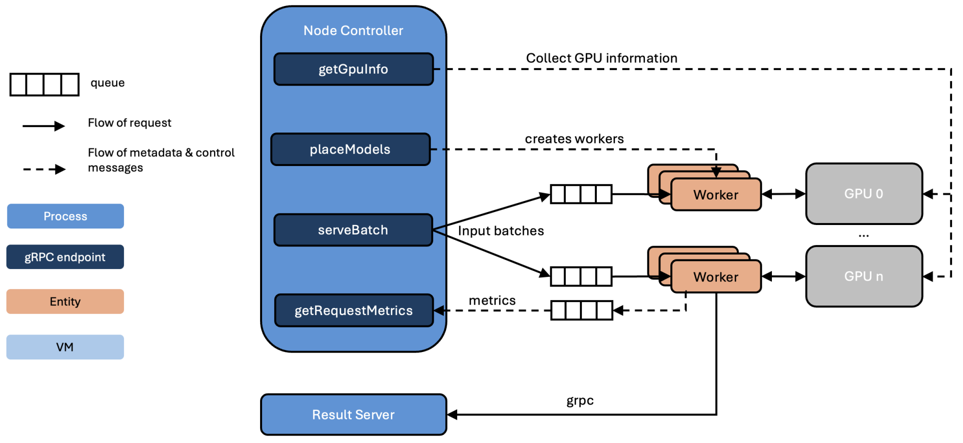


Figure 4.2: Node Controller

Figure 4.2 provides an overview of the node controller. As mentioned earlier in Section 4.1, the cluster is composed of at least one node, and each node is equipped with at least one GPU. Nodes can be viewed as local clusters themselves, each exposing an endpoint to receive and schedule work across its GPUs. The router introduced in Section 4.3.3 is the only client interacting with the node. For communication between the router and the node, each node runs a gRPC server [38], that we will refer to as the Node Controller from now on. Listing 4.5 shows the basic structure of this gRPC server, which is designed with a pool of threads to process multiple requests concurrently. The size of this thread pool can be configured by the user. Below, we will explore each of the server's endpoints in more detail.

Collecting GPU information

The `getGPUInfo` endpoint is used to collect details about the GPUs attached to a node. Listing 4.6 provides the pseudo code for this endpoint. As described in Subsection 4.3.1, the cluster controller needs this information before running the model placement policy. For each GPU attached to the node, the endpoint will create a `GPU` object as it was defined in section 4.2 and send it back to the cluster controller. The cluster controller will add the corresponding `node_id` to it. If the node has n GPUs attached to it,

```python
class NodeControllerGrpcServer():
    def __init__(self):
        self.metric_queue = init_shared_queue()
        self.metrics = dict()
        self.workers = dict()

    def getGPUInfo(self):
        # return information on all attached GPUs

    def placeModels(self, replicas: list[ModelPlacement]):
        # Update model placement and start workers

    def serveBatch(self, request: InferenceBatch):
        # forward request to worker

    def getRequestMetrics(self):
        # return collected metrics for all requests
```

Listing 4.5: NodeController gRPC Server

the `local_id` will range from 0 to $n-1$. The system is flexible enough to handle nodes with different types of GPUs.

Updating the model placement

Once the model placement policy has been executed by the cluster controller, it informs each node about the model replicas they will be handling. The `placeModels` endpoint is used for this purpose, as shown in Listing 4.7. This endpoint initializes new workers for each model replica and assigns them to a specific GPU. Additionally, the workers are provided with a shared queue for receiving inference requests and the node controller's shared queue for sending metrics back to it. The mapping from replicas to worker is maintained by the node controller in the `self.workers` map. Section 4.5 contains further information on the implementation of the workers.

Forwarding an inference batch

Once a request is submitted by a client, the router consolidates these requests into batches and forwards them to the nodes. The `serveBatch` endpoint, illustrated in Listing 4.8, serves as the entry point for processing these batches. Upon receiving a batch, the node controller records the timestamp and places the request in the appropriate worker's queue. The endpoint does not wait for the results of these requests but purely acts as a proxy. Prior to actually forwarding the request into the worker's queue, the endpoint first sleeps for some specific amount of time. We refer the reader to section 4.6 for the reasons and details behind this.

```python
class NodeControllerGrpcServer():
    ...

    def getGPUInfo(self) -> list[GPU]:
        gpus = []
        for each gpu attached to the node:
            gpus.append(
                GPU(
                    gpu.uid,
                    gpu.local_id,
                    gpu.gpu_type,
                    gpu.memory,
                    gpu.sm_count
                )
            )
        return gpus
```

Listing 4.6: `getGPUInfo` gRPC endpoint

```python
class NodeControllerGrpcServer():
    ...

    def placeModels(self, replicas: list[ModelPlacement]):
        for replica in replicas:
            input_queue = init_shared_queue()
            worker = start_new_worker(
                replica,
                input_queue,
                self.metric_queue
            )
            self.workers[replica.id] = worker
```

Listing 4.7: `placeModels` gRPC endpoint

Collecting Request Metrics

After all requests have been served, the `getRequestMetrics` endpoint, shown in Listing 4.9, is used by the Benchmark module to gather all collected metrics about the requests. The endpoint first initiates the termination of all workers on the node. As part of this termination process, the workers will communicate all their collected metrics back to the node controller using the `self.metrics_queue` that they were provided with upon initialization. Once all workers have terminated, the endpoint will then first gather the metrics of all the workers and send them back to the benchmark using gRPC streaming. Finally it streams its own collected metrics back to the benchmark module.

```
class NodeControllerGrpcServer():
    ...

    def serveBatch(self, request: InferenceBatch):
        # sleep for some time -> see section 4.6

        # forward request to worker input queue
        record_request_timestamp(self.metrics)
        worker = self.workers[request.replica_id]
        worker.input_queue.put(request)
```

Listing 4.8: `serveBatch` gRPC endpoint

```
class NodeControllerGrpcServer():
    ...

    def getRequestMetrics(self):
        terminate_all_workers(self.workers)

        # collect worker metrics
        worker_metrics = dict()
        while not self.metrics_queue.empty():
            worker_id, w_metrics = self.metrics_queue().get()
            worker_metrics[worker_id] = w_metrics

        # stream worker metrics back
        for w_id, w_metrics in worker_metrics:
            stream_metrics_back_to_benchmark(w_metrics)

        # stream node controller metrics back
        stream_metrics_back_to_benchmark(self.metrics)
```

Listing 4.9: `getRequestMetrics` gRPC endpoint

4.5 Worker Level

This section discusses the worker level of our ML inference cluster, providing an overview of its components, their interactions, and a detailed analysis of their functionality.

4.5.1 Overview

Figure 4.3 presents a detailed view of an individual worker. Each worker comprises two parallel processes: a `Dispatcher` and a `Result Processor`. These processes are launched by the node controller server through the `placeModel` endpoint (Section 4.4.1). The dispatcher is responsible for serving requests on the GPU, while the result processor handles the communication of results back to the result server at the cluster level (Section 4.3.4).

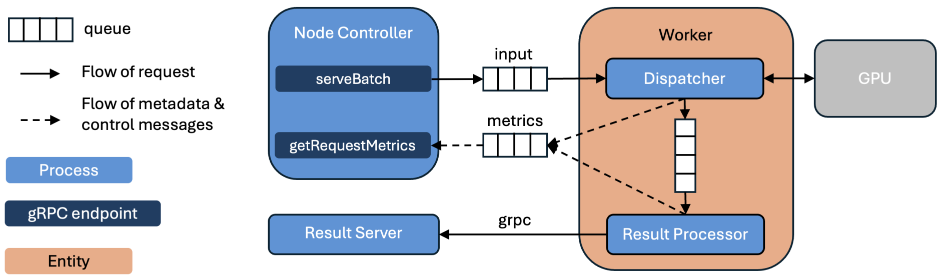


Figure 4.3: Worker

Both processes interact via a shared queue, enabling non-blocking operations. While the dispatcher submits results to the queue without waiting for the network transmission of the results to complete, the result processor manages the actual transmission asynchronously. The dispatcher can in the meantime proceed to serving the next request in the queue. We now explore the functionality of each process in greater detail.

4.5.2 Dispatcher

The dispatcher is responsible for performing inference tasks on the assigned GPU and its main sequence of action is provided by Listing 4.10. Key information for the dispatcher is encapsulated in the `replica` object:

- `replica.gpu.local_id` is the identifier of the GPU the dispatcher will use
- `replica.mps_share` is an optional setting that limits the number of Streaming Multiprocessors (SMs) available to the dispatcher
- `replica.model` is the model assigned to the dispatcher for inference

As each dispatcher serves a single model, the model is loaded onto the GPU once at the beginning. As long as the dispatcher remains active, it polls the `input_queue` for new requests. For each request, it generates input data based on the input shape provided, transfers the data to the GPU, executes the inference, and places the result onto the `result_queue` for further processing by the result processor. The dispatcher then continues to the next request without waiting for the result to be transmitted, thus overlapping computation with result transmission.

Requests in the `input_queue` arrive in batches, and are built by the router at the cluster level. The dispatcher processes these batches sequentially, without further grouping them into even larger batches. For each request, the dispatcher collects two metrics: the time at which a batch was dequeued and the inference latency. The former is used to capture the total

```python
def run_dispatcher(
    replica: ModelPlacement,
    input_queue,
    result_queue,
    metrics_queue
):
    metrics = dict()

    # set the optional MPS share
    if replica.mps_share is set:
        set_mps_share(replica.mps_share)

    model = load_model_to_gpu(replica.model,
       replica.gpu.local_id)

    while not terminated:
        req = get_next_request(input_queue)
        record_request_timestamp(metrics)

        # generate data from input shape -> simplification
        input_data = get_data(replica.model, req.input_shape)

        # copying and computing can be done in sep streams
        move_data_to_gpu(input_data, req.gpu.local_id)
        result = run_inference(model, input_data)
        record_inference_latency(metrics)

        # put result on queue and continue with next batch
        result_queue.put(result)

    # upon termination, send metrics to node controller
    metrics_queue.put(metrics)
```

Listing 4.10: `Dispatcher` process

time a request spends at the worker level, while the latter isolates the duration of the actual inference process, including only the data transfer from host to device, the computation on the device, and the transfer of the result back to the host. Whenever we talk about the inference latency we refer to the latter on.

Upon termination, the dispatcher sends all collected metrics back to the node controller.

Loading the model to the GPU

So far we have remained pretty vague on how a trained model ends up on the GPU of a worker. The `load_model_to_gpu` function takes care of this. Rather than transmitting the trained weights from the client all the way to the worker, a `Model` object, contained in the `replica.model` field, is sent.

This object includes the model name, which the worker uses to download the model and load it onto the GPU. The models used in our evaluation include vision models for object recognition from PyTorch [50] and language models for sentence classification from HuggingFace [16]. Further details about the models are provided in Section 6.1.2. The domain model can however easily be extended to include any API from which a model should be downloaded including private and closed source models.

Data Generation

Our observations, detailed in Section 4.6, reveal significant overhead when transmitting input data with each request from the client to the worker. This overhead is particularly pronounced with gRPC, especially when handling larger data volumes. Many of the models used in our evaluation are vision models with an input shape of $3 \times 224 \times 224$, resulting in a single request input size of $4 \cdot 3 \cdot 224 \cdot 224 = 602,122$ bytes (≈ 0.6 MB).

As batch sizes increase, the (de)serialization of the gRPC protocol buffer increasingly dominates the overall transmission latency, thereby limiting system throughput. Our measurements indicate that this overhead is specific to gRPC. Transmitting equivalent data volumes over raw TCP sockets yielded superior performance. Due to time constraints preventing a complete redefinition of the client-to-worker networking setup, we implemented a temporary solution: transmitting only the input shapes over the network and generating the actual data at the workers. Section 4.6 explains our approach to modeling network latency as if data were transmitted via raw TCP sockets, approximating a realistic setup. Future work should address this limitation by redefining the architecture to accommodate client-provided input data.

With data generation occurring at the workers, we must ensure efficiency to prevent domination of request processing time. Ideally, input data should be immediately available upon request dequeuing. Although each model replica has a desired batch size (accessible via `replica.batch_size`), we cannot assume that all batches dequeued from the `input_queue` will match this size. As explained in Section 4.3.3, the router employs a `MAX_WAIT_TIME` constant, after which the batch is dispatched to the node regardless of whether the desired batch size has been attained. To generate inputs with dynamic batch sizes, we pre-allocate a large batch (e.g., $512 \times 3 \times 224 \times 224$) in pinned memory [47]. Any call to the `get_data` function requesting a batch of shape $b \times 3 \times 224 \times 224$ returns the first $b \cdot 4 \cdot 3 \cdot 224 \cdot 224$ bytes from this pre-allocated tensor. We evaluated alternative options, such as regenerating random batches of the desired shape from scratch, but found them to introduce excessive overhead.

Setting the Multi-Process Service (MPS) Share

This section details the implications of running NVIDIA's Multi-Process Service (MPS) [24], particularly focusing on the effects of setting the `replica.mps_share`. While this functionality is specific to NVIDIA GPUs, our system is designed to be easily extensible to support other GPU architectures.

NVIDIA's MPS is a technology enabling concurrent sharing of a single GPU by multiple CUDA applications, primarily aimed at enhancing GPU utilization and overall system efficiency. From the Volta architecture onwards, NVIDIA introduced the capability to create soft partitions of the GPU by constraining the number of Streaming Multiprocessors (SMs) available to each process. It is important to note that while SM allocation can be constrained, underlying resources such as the L2 cache remain shared among all processes, distinguishing this approach from NVIDIA's Multi-Instance GPU (MIG) [23].

Our dispatchers operate with the MPS daemon running by default, facilitating concurrent GPU usage by multiple processes. Additionally, if the model placement policy requires constraining the number of SMs for each worker, it can do so by setting the `replica.mps_share`, a technique employed in works such as iGniter [81]. In such cases, the dispatcher process sets the `CUDA_MPS_ACTIVE_THREAD_PERCENTAGE` environment variable to limit available SMs. If this variable is not set, all SMs remain accessible to all processes. It is crucial to note that MPS is exclusively available on Linux systems and requires NVIDIA GPUs with a compute capability of 3.5 or higher.

Utilizing Streaming

Our dispatcher offers the option to use separate CUDA streams [22] for overlapping data copying and inference computation. When CUDA streaming is enabled, host-to-device data copying can be overlapped with kernel executions from previous requests. This is achieved by utilizing one CUDA stream dedicated to data copying and another for kernel execution. CUDA events are used to synchronize these streams, ensuring that a request's kernel execution starts only after its data has been copied to the GPU.

In the streaming scenario, the dispatcher process does not wait for the computation result to be copied back to the CPU before placing it on the `result_queue` and proceeding to the next request. Instead, it places an additional event on the queue that is set once the computation result is ready. The result process is responsible for waiting on this event before sending the computation result back, allowing the dispatcher to proceed to the next request and transmit data from host to device while the previous request's computation may still be ongoing. In non-streaming mode, the dispatcher always waits for the result to be available before placing it on the `result_queue`. It should be noted that the use of streaming necessitates the allocation of input data on the host in pinned memory [47].

Given these considerations, our system currently supports four distinct dispatcher configurations:

1. No `mps_share` setting and no streaming
2. No `mps_share` setting with streaming for overlapped data copying and computation
3. `mps_share` setting without streaming
4. `mps_share` setting with streaming for overlapped data copying and computation

4.5.3 Result Processor

```python
def run_dispatcher(result_queue, metrics_queue):
    metrics = dict()

    while not terminated:
        result = get_next_result(result_queue)

        # in case of streaming -> wait on result to be ready

        record_result_timestamp(metrics)
        send_result_to_result_server(result)

    # upon termination, send metrics to node controller
    metrics_queue.put(metrics)
```

Listing 4.11: `Result Processor` process

The result processor's functionality is straightforward, as illustrated in Listing 4.11. It operates continuously until terminated by the node controller, polling the `result_queue` and transmitting results back to the result server via gRPC. In the case of a streaming dispatcher, the result processor must first ensure result availability by waiting on the CUDA event placed on the queue by the dispatcher before retrieving and transmitting the result. As with the router, data transmission over the network introduces latency. To mitigate this overhead, the result processor uses multiple coroutines. This approach allows for concurrent processing: when a coroutine initiates the transmission of a computation result, it yields control back to the event loop. The event loop can then schedule another coroutine to proceed with sending the next result, even as the previous transmission may still be in progress. This asynchronous design effectively masks network latency, optimizing the overall throughput of the result transmission process.

4.6 Networking

In this section, we analyze the networking infrastructure and discuss the issues encountered during the project. We also present the simplifications applied to temporarily bypass these problems.

4.6.1 The choice of gRPC

As explained in Section 4.1, all components of the cluster, node, and worker-level systems can live on the same machine, allowing communication through shared memory. However in order to scale the cluster and accommodate larger GPU sets, we need to use separate machines for the nodes, which requires part of the communication to happen over the network.

We chose gRPC (Remote Procedure Calls) as the networking framework due to its ease of defining services via protocol buffers [13] and its language-agnostic nature. As a result, the node controller and the result server were implemented as gRPC servers.

The node controller's primary function, beyond receiving model placement information, is to act as a proxy, forwarding client requests to the workers. To this end, the node controller exposes the `serveBatch` endpoint, whose functionality is delineated in Listing 4.8. As our work progressed, this endpoint increasingly emerged as a bottleneck for the entire cluster, when transmitting larger batches along with their input data from the router to the nodes.

The observed network throughput failed to meet the expected bandwidth-based throughput, especially for vision models employed in object recognition tasks. Models such as `resnet50`, `vgg19` or `mobilenet_v2` [50] typically require inputs of shape $3 \times 224 \times 224$, translating to approximately 0.6MB of data when utilizing 32-bit floating point numbers. Common batch sizes in our system, such as 16 or 32, result in approximate data sizes of 9.6MB and 19.2MB, respectively. We observed that gRPC's performance began to degrade significantly for message sizes exceeding 1MB, with a marked decrease in throughput.

To further investigate, we compared the performance of gRPC, gRPC with client streaming, and raw TCP sockets in Python. We performed tests using two `n1-standard-16` VMs with `Intel Skylake` CPU platforms, located in the same Google Cloud Availability Zone. With `iperf3` [18], we measured an available bandwidth of 13.6 Gbps. The experiment involved sending an equal amount of raw bytes from one client to one server, comparing:

- **raw synchronous gRPC**: The whole payload is sent as a single message.

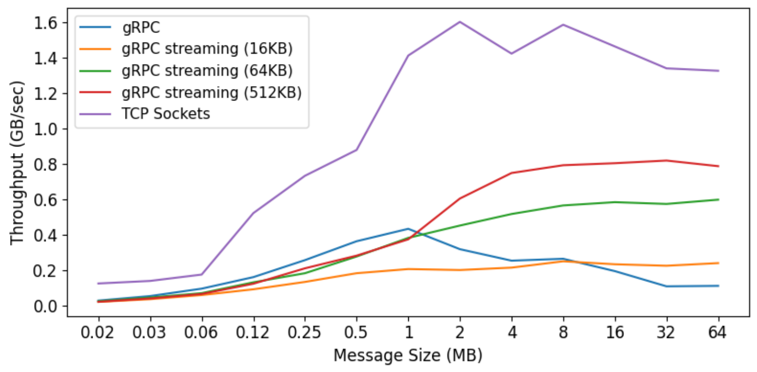


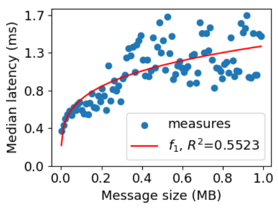
Figure 4.4: Comparing achieved throughput in GB/sec between gRPC, gRPC with streaming (chunk sizes 16KB, 64KB and 512KB) and raw TCP sockets

- **gRPC client streaming**: The whole payload is split into smaller chunks and send via streaming. We have conducted measurements with different chunk sizes.
- **raw TCP sockets**: The whole payload is sent using raw TCP.

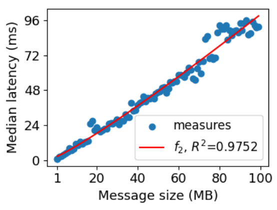
Figure 4.4 illustrates the results of these measurements. We conducted 20 consecutive runs for each message size, reporting the average throughput. Initially, throughput is low due to insufficient message sizes to saturate the network bandwidth. As message sizes increase, gRPC implementations demonstrate an inability to fully utilize the available bandwidth. While increasing chunk sizes in the client streaming scenario yields some improvement, only raw TCP sockets achieve bandwidth saturation. We attribute this performance degradation to the (de)serialization of protocol buffers at both the client and server ends.

4.6.2 Simulating the Network Transmission Latency

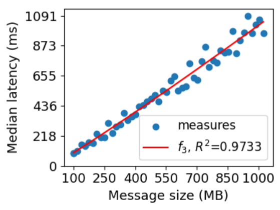
Our observations led us to conclude that while gRPC is suitable for transmitting messages with small payload sizes, it is suboptimal for frequent transmission of larger data volumes through the network. Raw TCP sockets would have been more appropriate for this purpose. Given that the primary objective of this work was to evaluate the impact of various placement policies on overall cluster throughput, it was crucial to prevent the network from becoming a bottleneck. Our preference was for GPU execution to be the limiting factor. Due to time constraints, a complete redesign of our network setup was not feasible. Consequently, we opted to generate input data at the workers and simulate the transmission time that would have been incurred had this data been sent from the client to the worker using raw TCP



(a) f_1



(b) f_2



(c) f_3

Figure 4.5: Measured median network transmission latency across 20 runs using raw TCP sockets in Python between two `n1-standard-16` VMs with the `Intel Skylake` CPU platform within the same Google Cloud Availability Zone. A different function f_i is fit for each of the three measurement intervals.

sockets.

To achieve this, we profiled the network transmission latency for messages of varying sizes from client to server, using these measurements to fit a function interpolating the expected network transmission latency for any payload size. Specifically, we utilized two `n1-standard-16` VMs with the `Intel Skylake` CPU platform within the same Google Cloud Availability Zone. Measurements conducted with `iperf3` revealed an average available bandwidth of 16.6 Gbps between the two VMs, with an average ping latency of 0.322 ms. We collected the following measurements:

- 100 payload sizes equally distant between 1KB and 1MB
- 100 payload sizes equally distant between 1MB and 100MB
- 50 payload sizes equally distant between 100MB and 1GB

For each payload size, we conducted 20 measurements and retained the median latency. For all three intervals i, we fit a function f_i that interpolates the expected median network transmission latency based on the message size. The function f_i is of the form

$$f_i(x) = a_i * x^{b_i} + c_i$$

The measured median latencies as well as the plotted functions f_1, f_2 and f_3 can be seen in figure 4.5.

Instead of transmitting the actual input data from client to worker, we send only the input shape. This shape allows us to determine the message size as if the actual input data had been transmitted over the network, enabling us to use our interpolated function to estimate the expected transmission latency. Upon receiving a message, the gRPC server uses the input shape attached to each request to determine the expected network transmission latency. Before processing the request, it induces a sleep period corresponding to this latency, thereby simulating the actual data transmission as if TCP sockets had been employed. This approach aims to maintain a realistic networking setup. It is important to note that this simulation is applied only to the `serveBatch` endpoint in the node controller server and the result server, as these are the sole components affected by the performance degradation on larger message sizes. All other communication continues to occur normally over gRPC without simulating network transmission latency.

To illustrate this process, consider a scenario where the router sends a batch of shape $16 \times 3 \times 224 \times 224$ for object recognition to a node. The router transmits only the shape dimensions (together with some other data) to the node, keeping the message size small. Upon reception, the node calculates the payload size of the input data, assuming 32-bit floating point numbers: $4 \cdot 16 \cdot 3 \cdot 224 \cdot 224 = 9'633'792$ bytes. Given this message size, the node employs the interpolated function f_2 to calculate the expected latency: $lat_{exp} = f_2(9'633'792) \approx 8.7$ ms. Before forwarding this batch to the corresponding worker, the thread in the node controller handling this request induces a sleep period of 8.7 ms. For sentence classification tasks using language models, this issue is less pronounced as the inputs are significantly smaller, typically ranging from 4 to 8 KB (depending on whether 32 or 64-bit integers are used) when employing a sequence length of 512 tokens and its corresponding attention mask. For messages of this size, we observe no substantial difference between using gRPC and raw TCP sockets but we still apply the same procedure as outlined above.

Future work should address this simplification by implementing a more appropriate networking setup that enables the transmission of input data from client to workers without introducing a network bottleneck.

4.7 Current limitations

This section highlights several limitations of our current system that should be addressed in future work.

4.7.1 Models are expected to fit on a single GPU

Our current system assumes that all model replicas can be accommodated on a single GPU, and does not support models whose size exceeds the capacity of a single GPU. To address this limitation, the `ModelPlacement` abstraction needs to be extended. One potential approach would be to introduce the concept of ranks for each worker involved in the model's inference process, along with the notion of a primary worker. For instance, Usher [76] implements this idea, utilizing DeepSpeed [7] to manage the underlying communication between all involved workers.

4.7.2 Lack of Support for Dynamic Placement Policy

The current system does not support a dynamic placement policy. The cluster controller executes the placement policy once at the beginning, and it remains static throughout the evaluation period. A more realistic setup should not assume that all models are available from the beginning, but rather that clients arrive and depart at various points during the evaluation. Furthermore, it should be possible to update the model placement dynamically based on the actual performance of the cluster. We have outlined how the result server (subsection 4.3.4) could be utilized to ingest feedback data back into the model placement policy. This would also necessitate additional functionality to dynamically update the current model placement. Workers unaffected by the updated placement should be able to continue serving without interruption.

4.7.3 Move clients to a separate machine

As discussed in section 4.3.6, the decision to co-locate clients with other cluster controller components on the same machine was a simplification made to quickly establish a functional cluster. Due to time constraints, we were unable to separate them onto distinct machines. We have already outlined potential solutions for future work section 4.3.6.

4.7.4 gRPC bottlenecks the overall throughput of the cluster

As analyzed in section 4.6, gRPC proved to be a suboptimal choice for transmitting large volumes of data at high rates through the cluster. While gRPC is well-suited for sending numerous control messages of smaller size throughout the cluster, an alternative network architecture needs to be implemented to facilitate the transmission of input data from clients to workers efficiently.

Chapter 5

Placement Policies

In this chapter, we introduce the model placement policies evaluated as part of this work. Section 5.1 introduces iGniter [81] and section 5.2 introduces Usher [76]. In section 5.3 we discuss alternative metrics to be used by the Usher placement policy. Finally in section 5.4 we present a modified version of Usher based on a Mixed Integer Linear Programming solver.

5.1 iGniter

This section examines iGniter [81], an interference-aware ML inference scheduler designed for spatially collocating multiple inference workloads on a single GPU. We have chosen iGniter as a baseline, given its ability to model interference created by colocated workloads on the same GPU. We outline the core concepts of iGniter in Section 5.1.1, followed by a description of its analytical model in Section 5.1.2. Section 5.1.3 describes iGniter's placement algorithm, Section 5.1.4 discusses the required profiling setup and we finally elaborate on the integration of iGniter into our system in Section 5.1.5.

5.1.1 Overview of iGniter

iGniter is built on the premise that temporal sharing of GPU resources among multiple workloads is insufficient to maximize GPU resource utilization and efficiency. Thus, spatial colocation on the GPU becomes necessary. However, as models are colocated on the same GPU, they begin to interfere with one another. iGniter introduces an analytical model that estimates the inference latency of a workload by accounting for the interference created by colocated models, thereby ensuring that the Service Level Objectives (SLO) of individual workloads can still be met. iGniter employs this analytical model to heuristically generate a model placement plan based on the following objectives:

Notation	Definition
t_{inf}^{ij}	Inference latency of workload i on GPU j
t_{load}^{i}	DNN inference data loading latency of workload i
$t_{feedback}^{i}$	DNN result feedback latency of workload i
t_{gpu}^{ij}	GPU execution latency of an inference workload i on GPU j
t_{sch}^{ij}	Scheduling delay of an inference workload i on GPU j
t_{act}^{ij}	GPU active time of an inference workload i on GPU j
r^{ij}	GPU resource allocation of an inference workload i on GPU j
f_j	Actual frequency of a GPU j
F	Maximum GPU Frequency

Table 5.1: iGniter Notation

1. It colocates models such that the increase in inference latency of colocated models due to interference is minimized while still guaranteeing the SLO.
2. It tries to limit the number of GPUs required to serve all workloads.

5.1.2 Analytical Model to predict the inference latency

We provide a high-level intuition of the analytical model introduced by iGniter and refer the reader to the original paper for more detailed information. Table 5.1 defines the notation for the analytical model that was in parts copied from the original paper.

The inference latency of a workload i on GPU j is given by equation 5.1. It represents the sum of the time required to load data from host to device, the time for computation on the GPU, and the time to copy results back from device to host.

$$t_{inf}^{ij} = t_{load}^{i} + t_{gpu}^{ij} + t_{feedback}^{i} \tag{5.1}$$

Both the loading time t_{load}^{i} and the feedback time $t_{feedback}^{i}$ depend on the size of the data that needs to be transmitted per request, the batch size and the bandwidth available on the PCIe bus.

$$t_{gpu}^{ij} = \frac{t_{sch}^{ij} + t_{act}^{ij}}{\frac{f_j}{F}} \tag{5.2}$$

iGniter's primary contribution lies in the calculation of t_{gpu}^{ij}, the time required for the kernels of a workload i to execute on a GPU j while potentially colocated with other workloads. The formula is provided by equation 5.2. iGniter identifies three potential sources of interference that will impact a workload's *gpu* latency.

1. t^{ij}_{sch} incorporates the scheduler overhead incurred by a workload's kernels as they are colocated with kernels of other workloads. iGniter models this delay as a linear function with respect to the number of colocated workloads. The authors assume that the GPU scheduler allocates kernels of different inference workloads in a round-robin manner. Consequently, as more workloads are colocated, the time for the next kernel of the same workload to be scheduled increases.

2. t^{ij}_{act} quantifies the time that GPU j actively executes a kernel from workload i. While setting MPS [24] shares prevents multiple workloads from contending for available Streaming Multiprocessors (SM), they still share underlying resources such as the L2 cache. The authors' measurements demonstrate that GPU active time exhibits an inverse relationship with the L2 cache hit ratio of the workload. A high hit ratio suggests low contention for L2 cache space among different workloads, resulting in low GPU active time. Conversely, a low hit ratio indicates severe contention at the L2 cache level, leading to longer active time.

3. Finally, the authors state that as the number of colocated workloads on a GPU increases, so does the power consumption. To prevent the GPU from exceeding its power cap, it automatically adjusts its frequency. The slowdown in GPU speed can be quantified by the ratio $\frac{f_j}{F}$, where f_j is the GPU's frequency under colocation and F denotes the maximum GPU frequency.

This model requires a number of hardware and workload specific coefficients that need to be acquired upfront by conducting some profiling on the workloads that are going to be served. We will address the profiling in Section 5.1.4 and refer the reader to the original paper as well as the open-sourced implementation of iGniter [17] for the remaining details.

5.1.3 A heuristic placement algorithm to minimize cost

With the analytical model established, iGniter addresses the following problem: Given an expected request arrival rate and a latency SLO, what batch size b_i and how much resources r^{ij} should be allocated to workload i running on GPU j to fully serve all requests within the SLO while minimizing the total cost. It is important to note that the GPU resource share r^{ij} refers to the MPS share. For instance, if $r^{ij} = 0.5$, workload i is allocated 50% of all SMs on GPU j.

iGniter's placement algorithm heuristically assigns workloads to GPUs and dynamically adjusts the GPU resources allocated to each workload as new workloads are added, ensuring continued fulfillment of the SLO. The algorithm proceeds as follows:

1. Workloads are sorted in descending order based on their r^i_{lower} value, which represents the minimum amount of resources a model requires to serve its workload when running alone on a GPU with a batch size b_i. Both r^i_{lower} and b_i are obtained through profiling (Section 5.1.4). Thus the more heavy and resource demanding workloads are placed first which is driven by the goal to reduce GPU fragmentation.

2. iGniter then greedily attempts to place workloads onto GPUs that may already be partially occupied by other workloads. It selects the GPU that, according to its analytical model, will result in the least increase of interference caused by the addition of the new workload.

3. For each GPU, the algorithm may incrementally increase the GPU resources assigned to each workload until the SLOs of all workloads can be met again. This step-wise increase is feasible as long as the sum of GPU resources allocated to all workloads on the GPU does not exceed 100%.

4. If no existing GPU can host the model while still guaranteeing the SLOs of the models it already hosts, a new GPU is added to the cluster.

5.1.4 Profiling

The implementation of iGniter is available as open-source software on GitHub [17]. It utilizes the NVIDIA Triton Inference server with the TensorRT runtime [53]. The repository also includes the necessary profiling scripts for collecting hardware and workload-specific coefficients required for the analytical model. These scripts are specifically designed for the TensorRT backend. As our system is based on PyTorch, we needed to modify portions of these scripts to ensure compatibility with our setup and rerun them for all models used in the evaluation. This section will first describe the original iGniter profiling script configuration, followed by a discussion of the modifications and challenges encountered during the adaptation process to our system.

iGniter profiling

In addition to hardware-specific coefficients such as GPU frequency, PCIe bandwidth, or GPU idle power consumption, the analytical model requires a set of workload-specific coefficients. These include, for example, the L2 cache activity of a model when running with various batch sizes and MPS shares, the power consumption of a model when run in isolation, and the GPU active time of a workload. To collect these coefficients for each model, iGniter proceeds as follows:

1. It first loads the model under consideration from PyTorch and converts it into ONNX format [33].
2. It then compiles and optimizes the models using the TensorRT command line wrapper `trtexec` [54].
3. Finally it runs the compiled model with `trtexec` as part of the TensorRT runtime environment and uses the NVIDIA Nsight Systems [28], NVIDIA Nsight Compute [26] and NVIDIA System Management Interface [29] profiler to collect the necessary coefficients.

It is worth noting that although the paper mentions the use of streaming by Triton inference servers to overlap data copying and kernel execution of independent requests, all profiling scripts published in the repository left the `--streams` flag in the `trtexec` command at its default value of 1. Consequently, we also refrained from using streaming during our profiling to maintain consistency with the original iGniter implementation.

iGniter profiling scripts adapted to PyTorch

This section outlines the key differences between our profiling scripts and the original ones. The primary distinction lies in the use of an alternative inference runtime environment, specifically PyTorch instead of TensorRT. We have developed a Python script that replicates the functionality of the `trtexec` command-line tool, providing an identical configuration interface to that used in the original scripts. For a given model and configuration, our script utilizes pre-generated data to apply a consistent load on the model. As in iGniter, the same NVIDIA profilers can be employed to collect the required metrics in the background. Unlike iGniter, we neither compile nor optimize the model, instead using its raw loaded version from PyTorch.

During certain profiling measurements, we encountered issues when using the NVIDIA Nsight Compute (*ncu*) profiler while the MPS daemon was running. We noticed that the *ncu* release notes [25] indicate that profiling with *ncu* while MPS is running is unsupported and may result in undefined behavior. With NVIDIA Driver versions ≥ 525, the profiling scripts simply crashed. However, using a Driver version of 470 did not result in crashes and reported the required coefficients. Nevertheless, we remain uncertain about the reliability of these measurements, given that the undefined behavior mentioned in the *ncu* release notes is not explicitly tied to a particular driver version. We have brought this limitation to the authors' attention. Consequently, we were compelled to run all profiling measurements on the older 470 Driver, and all reported evaluation results for iGniter are also based on the 470 Driver.

This constraint unfortunately prevented us from using the latest versions of several libraries. For instance, CUDA 12.x requires a minimum driver version of $\geq 525.60.13$ [4]. Thus, CUDA 11.8 is the highest version we could use for iGniter. We attempted to build PyTorch from source and link it against a local CUDA 12.4 toolkit with the CUDA Forward Compatibility Package [5] installed, providing support for running CUDA 12.x on a 470 Driver. However, measurements revealed that this compiled version of PyTorch suffered from severe performance degradation compared to PyTorch compiled with CUDA 11.8. Consequently, we decided to use CUDA 11.8 with the 470 Driver. The exact setup for iGniter will be detailed in Chapter 6.

The original iGniter scripts contain several hardcoded constants that appear to be tightly coupled to the models evaluated in the paper. Some of these constants did not generalize well to the models we used additionally. During profiling, we occasionally had to adjust some of these constants for specific models to ensure the scripts successfully returned the coefficients.

5.1.5 Integration of iGniter into our system

Apart from adjusting the profiling scripts, the integration of the iGniter placement policy into our system proved to be relatively straightforward. We were able to incorporate the entire placement algorithm into our system with minor adjustments to adhere to the interface introduced in section 4.3.2. This validates our system's ability to abstract the underlying placement and expose a simple API. Note that iGniter also assumes the target request submission rate and the SLOs for each model to be known upfront and available to the placement policy.

As we conducted our evaluation in a fixed-size GPU cluster, we modified iGniter to function as a best-effort policy. Once the complete placement plan by iGniter is available, we greedily assign the placement as returned by the algorithm to GPUs until either all workloads are placed or the available resources are exhausted.

For the actual inference execution, we will choose a Worker, whose dispatcher process both sets the MPS share and uses CUDA streaming to overlap the data copying and computation of independent requests on the GPU. We refer the reader back to Section 4.5.2 for the details on this.

5.2 Usher

This section provides a detailed description of the Usher paper [76]. Usher is designed to holistically mitigate interference for resource optimized ML inference. As Usher outperforms GPUlet [60], AlpaServe [71], and Shepherd [82], works we had also considered as potential baselines, we considered Usher to be an optimal baseline for our research. Consequently, we have invested substantial time in comprehensively understanding and evaluating Usher.

It is worth noting that at the time of writing, only a small, non-functional portion of the Usher repository was open-sourced on GitHub [56]. We reached out to the authors with a series of questions regarding implementation details and requested full access to the code. However, due to time constraints, the authors have not yet been able to respond to our request but have assured us that they will do so as soon as possible. As a result, we have implemented the Usher placement policy from scratch based on our understanding of the paper and the partially published code. We will outline the main ideas and components of the Usher framework in Section 5.2.1 and provide a detailed explanation of our implementation along with the challenges that we have encountered in Section 5.2.2.

5.2.1 Overview of Usher

Usher is an ML inference scheduler designed to heuristically co-locate workloads on GPUs while mitigating interference. The Usher tool can operate in two distinct settings:

1. **Non-fixed sized cluster**: Given the target submission rate and the Service Level Objectives (SLOs) of each model to be served, Usher utilizes as many GPUs as necessary to fulfill the entire workload of all models within their respective SLOs. Among all placement scenarios that meet these requirements, Usher selects the one with minimal cost.
2. **Fixed-sized cluster**: Given a limited number of available GPUs, Usher's objective shifts to maximizing the cluster's overall goodput. In this setting, Usher aims to serve as many requests as possible without violating their SLOs using the limited number of GPUs available. It is important to note that in this setting, Usher provides no guarantees on whether the workloads of all models can be served.

The following paragraphs will describe the main steps of the Usher tool in achieving these goals.

Estimation of Model Resource Requirements

To determine whether to co-locate two models on the same GPU, Usher requires each model's resource requirements in terms of both memory ($Mreq$) and computation ($Creq$). Rather than undertaking the costly and time-intensive effort of profiling these requirements for each model individually, Usher proposes estimating them using pre-trained regression models, thus avoiding the need to execute the workloads upfront. Usher estimates the compute and memory requirements of individual kernels and then derives the overall model requirements from these estimates. Usher employs three stacked regression models for this purpose, each taking as input the batch size, the size of the kernel's parameter weight tensor and input tensor, the number of floating point operations conducted by the kernel, and the GPU type:

1. The **Creq-Regressor** estimates the computational requirement of the kernel. Usher uses the `achieved_occupancy` [21] metric to quantify the compute requirements of a kernel. The `achieved_occupancy` represents the ratio between the number of active warps during a kernel's execution and the maximum number of warps that can be active, across all SMs in a GPU.
2. The **Mreq-Regressor** estimates the memory requirement of the kernel. Usher uses the `dram_utilization` metric to quantify the memory requirements of a kernel. According to the NVIDIA Nsight Compute (*ncu*) documentation [19], the `dram_utilization` metric is part of the older NVIDIA *nvprof* [32] profiler, and mapped to the equivalent `dram__throughput.avg.pct_of_peak_sustained_elapsed` metric in the *ncu* profiler. To our understanding, this metric measures DRAM bandwidth utilization rather than DRAM capacity utilization, which appears to contradict the remainder of the paper that seems to consistently refer to DRAM capacity when discussing $Mreq$.
3. The **Time-Regressor** estimates the execution time of a kernel.

Using these regression models, Usher estimates a model's $Creq$ and $Mreq$ for a given batch size as follows:

1. Usher begins by decomposing the model's operator computation graph, replacing each operator node in the graph with the sequence of kernels it invokes.
2. For each kernel in the graph, Usher utilizes the Creq-Regressor and the Mreq-Regressor to calculate the kernel's compute and memory requirements.
3. Using the Time-Regressor and the dependency ordering of kernels in the computation graph, Usher determines the start and end time of each kernel as an offset from the start of the very first kernel in the graph. If the computation graph has parallel branches, indicating that kernels might execute concurrently as part of different streams, Usher can leverage the timing information to determine whether the computation of two kernels could potentially overlap.

Usher calculates the $Creq_{m,b}$ of a model m with batch size b based on equation 5.3 as the maximum sum of $Creq$ of any set of overlapping kernels.

$$Creq_{m,b} = \max_{S \in sets\_of\_overlapping\_kernels} \left(\sum_{i \in S} Creq_i \right) \tag{5.3}$$

The $Mreq_{m,b}$ for a model m with batch size b is calculated based on equation 5.4. It is the sum of the size of the model's weights and the maximum sum of $Mreq$ of any set of overlapping kernels.

$$Mreq_{m,b} = model\_size + \max_{S \in sets\_of\_overlapping\_kernels} \left(\sum_{i \in S} Mreq_i \right) \tag{5.4}$$

Grouping compute and memory heavy models

Once the $Mreq$ and $Creq$ values for each model and batch size are determined, Usher groups the models into smaller, nearly equal-sized groups such that the sum of the $Creq$ and the sum of the $Mreq$ of all models in each group are approximately equal, i.e., $\sum_i Creq_i \approx \sum_i Mreq_i$. The motivation behind forming smaller groups with balanced compute and memory requirements is to increase the likelihood of colocating models with complementary resource needs and to reduce fragmentation. To achieve this, Usher uses a variant of the K-means clustering algorithm, which proceeds in several rounds as follows:

1. Initially, each model starts as its own group.
2. Usher calculates the distance D between any two groups as $D = |\sum_i Creq_i - \sum_i Mreq_i|$, where i iterates over the models of both groups.
3. It groups two models such that D is minimized.
4. Each new group is considered as a new element, and the algorithm repeats from step 2 until the desired group size is reached.

As the clustering algorithm has not been open-sourced by the authors, some ambiguities and confusion remain. Firstly, from the paper's description, we understand that in each round, pairs of models are merged into a group, equivalent to creating a maximum cardinality matching between the groups. It remains unclear how the authors have managed to control the group size in each iteration using a k-means based algorithm. Not limiting the group size could lead to unbalanced group sizes with k-means.

Additionally, the paper states that two models should be merged into the same group such that the distance D is minimized. This statement is ambiguous to our understanding, as it is not clear whether the distance D refers to the distance of a particular group or to the sum of distances of all groups. This could translate to both a local or a global objective criteria in each round:

- **Local objective**: Minimize the distance of each group individually, assuming some underlying greedy order.

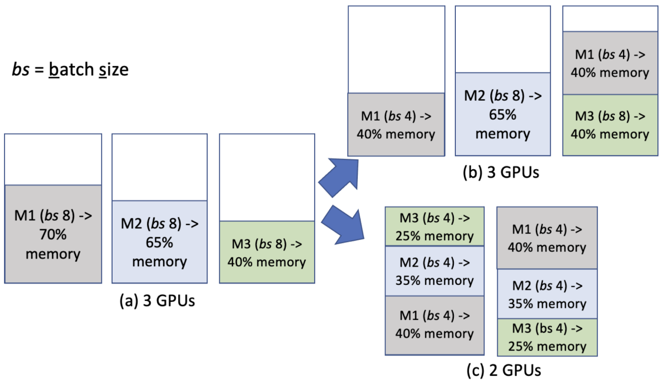


Figure 5.1: Concept of **model replicas** in Usher. The figure is taken from the Usher [76] paper.

- **Global objective**: Minimize the sum of distances $min(\sum_i D_i)$ of all groups.

The local objective assumes there is an underlying ordering in which the individual groups are visited, while the global objective is independent of any underlying order of visitation.

In Section 5.2.2, we present our choice of implementation for this grouping algorithm. Our implementation uses the global objective criteria in each round and ensures that no more than two models end up in the same group during a same round, utilizing a min-weighted maximum cardinality matching algorithm.

Heuristic Placement of Model Replicas

Once the groups are known, Usher proceeds to heuristically place one group after another onto the GPUs. The authors do not specify the order in which the groups are placed. When placing the models of a group, Usher first attempts to utilize any remaining resources from GPUs already hosting models from previously placed models before initializing new GPUs.

For the placement of models onto GPUs, Usher introduces the concept of a **model replica**. This concept is illustrated in Figure 5.1. While a certain batch size for a model might be able to fully satisfy a model's workload, this approach may result in a high degree of fragmentation. Instead, it could be more efficient to replicate the model multiple times with a smaller batch size, thereby reducing the resource requirements of each replica and consequently reducing fragmentation.

At the beginning of placing a group, Usher generates the set of all possible configurations for the models within the group. A configuration consists of a `(rep_factor, batch_size)` tuple for each model, specifying its replication degree and the corresponding batch size. The batch size for a model remains constant across all its replicas. Usher then heuristically attempts to identify the optimal configuration based on its optimization criteria of maximizing goodput and minimizing cost. Usher defines the goodput of a model as the ratio of the model's batch size to its expected time (including in-queue wait time) to serve the batch. However, neither the paper nor the open-sourced code provide details on how this in-queue wait time is modeled. In chapter 6, we will demonstrate that modeling the queuing delay is crucial for accurately estimating the goodput.

For each configuration, Usher first separates models into three groups. A model is considered memory heavy if $\frac{Mreq}{Creq} \geq 1.2$. It is considered compute heavy if $\frac{Creq}{Mreq} \geq 1.2$ or neutral otherwise. Each group is then sorted in decreasing order with respect to their total resource requirements $Rreq = Mreq + Creq$ for the batch size under consideration. It then places the models by alternating between memory and compute heavy models one after another onto the GPUs, first utilizing GPUs already used by the same group, then GPUs used by other groups with spare capacity, and finally adding GPUs to the cluster until the load of a model is served or no more GPUs are available. The neutral models are placed last.

We will demonstrate in Section 5.4 that both the order of placing groups and the greedy order of placing models within a group can lead to sub-optimal goodput in Usher.

Operator Graph Merging

The authors of Usher present an additional contribution in their paper, which involves merging operators of different computation graphs executing on the same GPU with similar weight matrices to reduce interference in the GPU cache. As our focus is on evaluating Usher's placement policy, we will not delve further into this topic. Interested readers are encouraged to consult the original paper for more information on this aspect.

5.2.2 Implementation of Usher

Given that only parts of the Usher codebase have been open-sourced at the time of writing, we have decided to implement Usher from scratch based on our understanding. As previously highlighted, numerous ambiguities remain unresolved and the decisions taken on crucial aspects of the algorithm are based on our best understanding of the paper and the published code. Notably, we encountered contradictions between the code and the paper (e.g. in the formula for estimating a model's goodput). We have aligned our implementation with the descriptions provided in the paper. Consequently, it remains unclear how accurate our implementation is compared to the original Usher, something the reader should keep in mind for the evaluation section. We will describe and defend our most important implementation choices below and invite the reader to review our code for the complete implementation.

Profiling Model Resource Requirements

As explained in Section 5.2.1, the Usher placement algorithm requires the memory and compute requirements of a model for various batch sizes. Unfortunately, none of the three regression models for estimating the *Mreq*, *Creq*, or latency of a kernel is available, and limited information is shared about their implementation and training process. Consequently, we have opted to explicitly profile the resource requirements for all models and batch sizes under consideration. This approach allows us to replace the three regression models with an oracle that can accurately predict the *Mreq* and *Creq* based on the profiled results.

Usher states that they conducted their profiling measurements for training the regression models using the NVIDIA Nsight Systems (*nsys*) [28] profiler with the `print-gpu-trace` option enabled. They utilized the `achieved_occupancy` for *Creq* and `dram_utilization` for *Mreq*. However, we believe that the authors may have used the *nvprof* profiler instead, as these terms are exclusive to *nvprof*, and `achieved_occupancy` is not even available in *nsys*. The *nvprof* profiler will be deprecated in a future CUDA release and has been replaced by the *nsys* profiler for system-wide metrics and the *ncu* for kernel-granularity metrics in the meantime.

For our oracle, we have decided to use combined measurements from the more up to date *nsys* and *ncu* profilers. Specifically, for each model and batch size, we conduct two profiling runs:

- Using *nsys* to collect the GPU trace with the `--report cuda_gpu_trace` flag
- Using *ncu* to collect the `sm__warps_active.avg.pct_of_peak_sustained_active` metric, which is equivalent to `achieved_occupancy` in *nvprof* [19]

By analyzing the GPU trace, we can identify concurrently executing kernels and, by matching this data to the *ncu* profiler outputs, we can aggregate the *Creq* values of the concurrently executing kernels to finally obtain the overall *Creq* for the entire model. We match the kernels between the *nsys* GPU trace and the *ncu* output based on their names, order and launch configuration.

For the *Mreq*, we have decided not to use the `dram_utilization` or its *ncu* equivalent `dram__throughput.avg.pct_of_peak_sustained_elapsed` metric. This decision is based on the paper's consistent reference to the *Mreq* metric as the maximum capacity required by the model during execution, and we believe that DRAM bandwidth utilization is not the appropriate metric for this purpose. Instead, given our use of PyTorch as the runtime environment, we use `torch.cuda.max_memory_reserved` [42], which provides the maximum amount of memory held by the Torch caching allocator at any point during model execution. This includes both the memory required to store model weights and the maximum memory capacity needed for intermediate computations of any set of concurrently executing kernels within the model. All profiled measures can be found in table A.1 of the Appendix.

Minimum weighted maximum cardinality matching for grouping

We have previously discussed the ambiguities about the model grouping in Usher in Section 5.2.1. It remains unclear how Usher has modified the k-means algorithm for their purpose. Notably, we are uncertain how the group size in each iteration of the algorithm is bounded such that no more than two models can be merged into the same group in each iteration. Additionally, the question of whether to optimize globally or locally within a round persists.

Instead of creating the groups based on a variant of k-means, we opted for a round-based minimum-weight maximum cardinality matching using the `min_weight_matching` algorithm from the networkx library [20]. The algorithm proceeds as follows:

1. Each model starts as its own group.
2. Calculate the pairwise distance between any two groups as defined by Usher.
3. Create a Graph with one node per group. Add an edge between any pair of groups and set its weight equal to the distance between the two groups.
4. Run the `networkx.min_weight_matching` algorithm.
5. Merge any two groups that have been matched to each other and repeat from step 2 until the desired group size has been reached.

The matching part of the algorithm ensures that in each round, no more than two models end up in the same group, and the maximum cardinality ensures that there can be at most one unmatched group at the end if the number of groups was odd. The minimum weight property ensures that we minimize the sum of all distances, corresponding to the global objective. It doesn't guarantee that the sum within a group is minimized. If the maximum desired group size is n, the algorithm requires $\lfloor \log_2(n) \rfloor$ rounds.

Choosing the right replication factor

The set of possible batch sizes for a model is given in the Usher paper as {4,8,16,32,64,128}. The situation for the replication factor is somewhat more ambiguous. Usher states that for each model m, the replication factor is chosen from the set $\{i \cdot c_m^l\}$, where $i \in \{1, 2, ..., 6\}$ and c_m^l is the minimum number of GPUs required to complete all of the model's m requests within the SLO when using the highest possible batch size. Usher also states that the maximum replication factor of a model to choose from is upper bounded by the total number of GPUs present in the cluster. Assuming a fixed-size cluster of 4 GPUs and a model m with $c_m^l = 5$, the model's replication factors to choose from would be $\{5, 10, 15, 20, 25, 30\}$. Due to the upper bound of 4 GPUs, the set of replication factors would be empty.

It remains unclear in the original Usher implementation whether such a model should not be served at all or be assigned a maximum replication factor of 4. We have decided in such instances to not serve the model at all. We will propose a modified version of Usher in Section 5.4 that will ignore the c_m^l multiplier.

PyTorch for inference execution

The Usher paper specifies the use of TensorFlow for inference execution. In contrast, we will be using PyTorch for our implementation and will not apply any optimizations or compile the models in any way. Similar to Usher, we will set the MPS share at the worker level based on its $Creq$.

5.3 Usher extended with different metrics

This section presents some of our own ideas to plug alternative metrics into the Usher placement policy.

5.3.1 Weighted Average Achieved Occupancy

Our profiling results reveal that the `achieved_occupancy`, suggested as the metric to quantify compute requirements ($Creq$), is remarkably high for all models and batch sizes profiled. This high occupancy is to such an extent that no two models can be co-located, regardless of batch size. The left plot in Figure 5.2 illustrates the $Mreq$ on the x-axis, measured using `torch.cuda.max_memory_reserved` [42], and the `achieved_occupancy` measured using *ncu* on the y-axis for a mixture of vision and language models [16,50]. Each model is represented by multiple points, one for each batch size. Precise values for each model and batch size can be found in Table A.1 of the Appendix.

The plot demonstrates that while the $Mreq$ for most models and batch sizes is below 50%, the achieved occupancy, with one exception for `alexnet`,

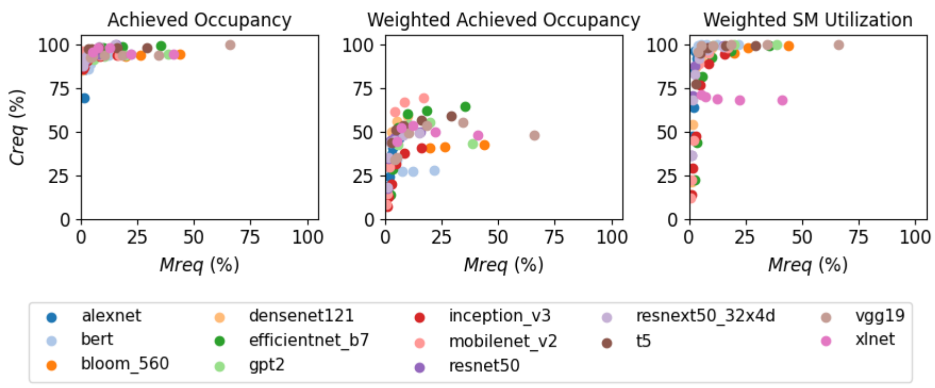
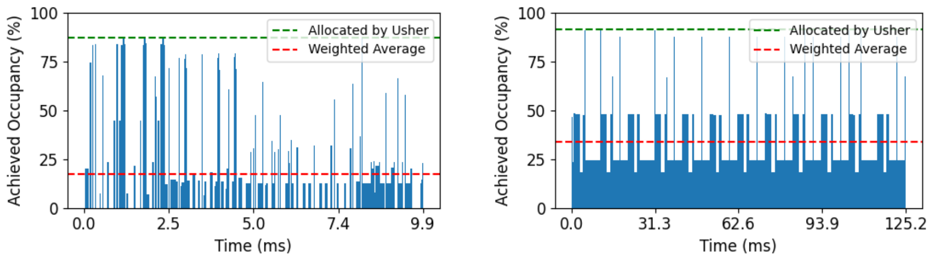


Figure 5.2: Three different *Creq* metrics considered for Usher measured on a NVIDIA V100 GPU in Google Cloud Platform. There is one point per model per batch size. The *Mreq* metric corresponds to `torch.cuda.max_memory_reserved` [42]



(a) `resnet50` with batch size 4

(b) `bert` with batch size 16

Figure 5.3: Achieved Occupancy measured for one iteration of object recognition and text classification run on a NVIDIA V100.

consistently exceeds 80%. Consequently, no two models will be co-located, as their sum in *Creq* would always exceed 100%, preventing Usher from co-locating them. With the exception of `alexnet`, `vgg19`, `bloom_560`, `t5` and `xlnet`, all other models have also been evaluated in the Usher paper. Given these measurements, we are uncertain how to reproduce co-location scenarios reported in the paper, such as `efficientnet_b7` with `bert` (also used for text classification).

Usher uses the maximum achieved occupancy across any set of overlapping kernels as the model's *Creq*. Instead we have analyzed how the achieved occupancy looks like for individual kernels an how it compares to the maximum allocated by Usher. Figure 5.3 presents two examples: `resnet50` with a batch size of 4 used for object recognition, and `bert` with a batch size of 16 used for text classification. Both plots measure a single batch execution and report the achieved occupancy over time. We have merged the metrics obtained through *ncu* with the CUDA GPU trace obtained through *nsys*. Each bar in the plot represents an interval. The width of the bar represents the interval length, and the height corresponds to the sum of achieved occupancies of all kernels whose execution overlaps with the interval. Interval boundaries correspond to the start and end times of all kernels within the model.

Both figures demonstrate that for most of the time, the achieved occupancy is significantly below the maximum reached and allocated by Usher, suggesting that substantial computational resources might remain unused for extended periods. We have added the weighted average to the plot, calculated as the sum of the interval lengths multiplied by their achieved occupancy, divided by the total execution time. The weighted average achieved occupancy is considerably below the maximum achieved occupancy and could be a better metric to quantify how much resources are on average required by a model over the whole execution time.

The middle plot in Figure 5.2 shows the weighted average achieved occupancy for the same models and batch sizes as in the left plot. We observe that using the weighted average achieved occupancy creates more potential for co-location from a purely arithmetic perspective, as most models and batch sizes now show a *Creq* between 25% to 50%. Note that we have not altered the *Mreq* value. Points from the left to the middle plot only move along the y-axis. The weighted average achieved occupancy for all models and batch sizes can be found in Table A.1 of the Appendix.

Based on these insights, we have created a slight modification of Usher that maintains the exact same algorithm but considers the weighted average achieved occupancy as the *Creq* instead of the maximum achieved occupancy.

5.3.2 Weighted SM Utilization

Building upon the idea of reusing the Usher algorithm with different metrics for the *Creq*, this section introduces the **SM Utilization**. The SM utilization for a kernel is defined as the ratio between the number of Streaming Multiprocessors (SMs) a kernel requires based on its launch configuration and the number of SMs available on the GPU. This value is capped at 100%. If the SM utilization exceeds 100%, it indicates that the blocks of the kernel will be scheduled in consecutive rounds. The SM utilization is mathematically expressed as:

$$sm\_utilization = min\left(100, \frac{num\_sms\_needed}{num\_sms\_on\_gpu}\right)$$

The following paragraphs describe the method for calculating the number of SMs needed (*num_sms_needed*). Using the *nsys* profiler, we can obtain the CUDA GPU trace for each kernel of a model, which provides the launch configuration of the kernel, including the dimensions of the grid size and the block size [6]. The total number of thread blocks is calculated as the product of the grid dimensions:

$$num\_thread\_blocks = grid_x \cdot grid_y \cdot grid_z$$

Thread blocks in GPU computing are scheduled onto Streaming Multiprocessors (SMs) and are never split across multiple SMs. Multiple blocks can be scheduled onto the same SM. The number of blocks that can be executed concurrently on an SM is limited by several factors and depends on the GPU's capabilities [21, 30].

Factors Limiting Concurrent Block Execution

1. **Maximum number of threads per SM**: Each GPU has a maximum number of threads per SM that can execute concurrently. This limits the number of blocks that can execute on the SM. The maximum threads per SM is obtained from the GPU's technical specifications, and the number of threads per block is the product of the block dimensions.

$$bound\_max\_threads = \left\lfloor \frac{max\_threads\_per\_sm}{block_x \cdot block_y \cdot block_z} \right\rfloor$$

2. **Maximum shared memory per SM**: Each SM has a maximum amount of shared memory distributed across all threads running on the SM. The shared memory required by a block is the sum of static shared memory (used for `__shared__` variables in the kernel) and dynamic shared memory specified in the kernel launch configuration.

$$bound\_shared\_mem = \left\lfloor \frac{max\_shared\_mem\_per\_sm}{stat\_shared\_mem + dyn\_shared\_mem} \right\rfloor$$

3. **Maximum number of registers per SM**: Each SM has a maximum number of registers available to be shared across all executing threads. This value is obtained from the technical specifications, while the number of registers used per thread is obtained from the GPU trace.

$$bound\_registers = \left\lfloor \frac{max\_regs\_per\_sm}{block_x \cdot block_y \cdot block_z \cdot regs\_per\_thread} \right\rfloor$$

4. **Maximum number of blocks per SM**: Each SM has a maximum number of blocks *bound_blocks* that it can execute concurrently, as specified in the technical specifications.

The maximum number of blocks *num_blocks_per_sm* that can be executed concurrently on an SM is the minimum value across all four bounds above. Finally the number of SMs needed *num_sms_needed* for running a kernel based on its launch configuration is calculated as

$$num\_sms\_needed = \left\lceil \frac{num\_thread\_blocks}{num\_blocks\_per\_sm} \right\rceil$$

Motivation for Using Weighted SM utilization

The SM utilization can be used to quantify whether there are enough SMs on the GPU to execute multiple kernels from different models. For example, on an NVIDIA V100 GPU with 80 SMs:

- If kernel 1 requires 50 SMs and kernel 2 requires 20 SMs, both kernels' blocks can be launched on distinct sets of SMs.[1]
- If however kernel 2 requires 40 SMs, the total sum of required SMs (90) exceeds the available 80 SMs. This results in interference at the SM level, as some blocks must wait for SMs to become available.

It's important to note that SM utilization does not account for interference at lower levels of the GPU, such as the L2 cache for example.

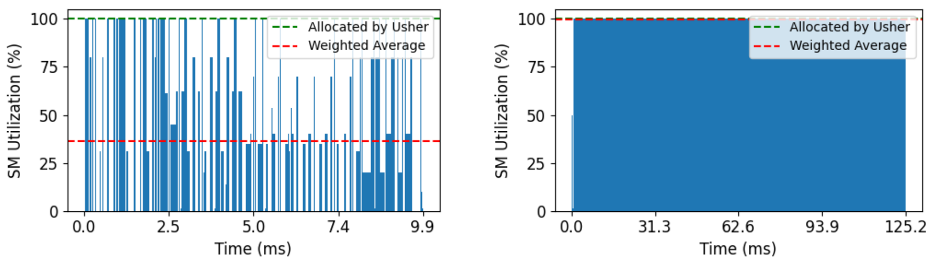
Observations on SM Utilization

We have calculated the SM utilization for all kernels across various models and batch sizes under consideration. Our analysis reveals that when using the maximum SM utilization of any kernel as a model's computational requirement (*Creq*), the *Creq* would be 100% for all models and batches examined. Figure 5.4 illustrates the SM utilization again for `resnet50` with a batch size of 4 and `bert` with a batch size of 16. These values are derived from the same profiling measures used in our previous analysis of achieved occupancy and its weighted average counterpart in Figure 5.3.

For `resnet50`, we observe that using the maximum SM utilization as the model's *Creq* would result in some SMs remaining unused for certain kernels. This suggests potential inefficiencies in resource allocation when relying solely on maximum SM utilization as a metric. In contrast, for `bert`, we note that the weighted average SM utilization and maximum SM utilization essentially overlap. This convergence occurs because all kernels invoked in `bert` consistently require all available SMs on the GPU, indicating a high and constant demand for computational resources.

A comparison between Figures 5.4b and 5.3b yields interesting insights. We have employed two distinct metrics to quantify the computational requirements for the same model and batch size. Notably, while the weighted

[1]The exact placement of kernel blocks to the SMs depends on the GPU block scheduler.



(a) `resnet50` with batch size 4 (b) `bert` with batch size 16

Figure 5.4: SM Utilization measured for one iteration of object recognition and text classification run on a NVIDIA V100.

average achieved occupancy suggests that `bert` could potentially be collocated with another model, the weighted SM utilization metric indicates that no model or batch size could be collocated with `bert` under the Usher colocation criteria. This highlights the importance of choosing appropriate metrics for assessing resource utilization and making informed decisions about model collocation.

Finally the right most plot of figure 5.2 shows the weighted SM utilization for all profiled models and batch sizes. While a number of models, particularly language models, have a high weighted SM utilization excluding colocation, some models, especially vision models with small batch sizes show potential for colocation. Table A.1 of the Appendix contains the weighted SM utilization of all profiled models and batch sizes. We will compare the weighted achieved occupancy and weighted SM utilization in greater detail in chapter 6.

5.4 Usher extended with Mixed Integer Linear Programming

In this section we analyze some issues related to the Usher placement policy that are specific to the fixed size cluster and do not appear in the non-fixed size cluster. Section 5.4.1 analyzes Usher's placement order of models and Section 5.4.2 discusses its replication factor multiplier. Finally we present a Mixed Integer Linear Programming approach in Section 5.4.3 that addresses these issues.

5.4.1 Problematic Placement Order within and Across Groups

As discussed in Section 5.2.1, the order in which models are placed within groups or across groups can significantly impact Usher's goal of maximizing the cluster's overall goodput. In this section, we examine specific scenarios

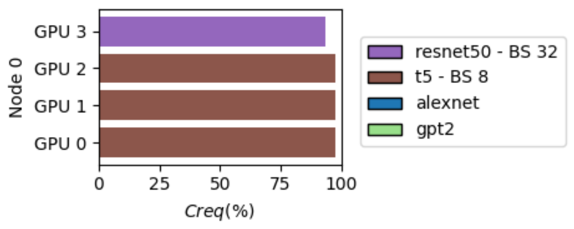


Figure 5.5: Model placement obtained from Usher for `alexnet`, `gpt2`, `resnet50` and `t5` each having an expected request arrival rate of 400 requests per seconds and an SLO of 200ms. The x-axis shows the achieved occupancy.

that demonstrate suboptimal outcomes due to Usher's placement heuristics.

Problematic Heuristic Order within a Group

Consider a scenario with two vision models [50], `resnet50` and `alexnet`, and two language models [16], `t5` and `gpt2`. Each model has an expected request arrival rate of 400 requests per second (req/s) and a Service Level Objective (SLO) of 200 ms. The cluster is fixed and has one node with 4 GPUs attached to it. The maximum group size is 4, as specified in the Usher paper, i.e the clustering algorithm will return one group containing all 4 models. Table 5.2 contains the required profiled metrics. We use the achieved occupancy as *Creq*. As a reminder the expected goodput is calculated as the ratio between the batch size and the time it takes to execute it.

The model placement obtained by our implementation of Usher is displayed in Figure 5.5. The expected goodput for this placement is

$$min(400, 3 \cdot 137.38) + min(400, 1067.13) = 800$$

requests per second calculated as the minimum of the models' expected request arrival rates and their respective replicated goodputs.

However, a more optimal placement in terms of maximizing the total goodput could have been achieved by placing one replica of `alexnet` with a batch size of 4, one replica of `resnet50` with a batch size of 4, and two replicas of `t5` with a batch size of 16. This would have resulted in an expected goodput of 1092.04 req/s. The reason that Usher cannot find this placement has to do with the order in which the models are placed onto the GPUs. An ordering in which both `alexnet` and `resnet50` appear before `t5` is not possible according to Usher. The reasoning is as follows:

1. Usher first discards all batch sizes whose latency violates the 200 ms SLO.

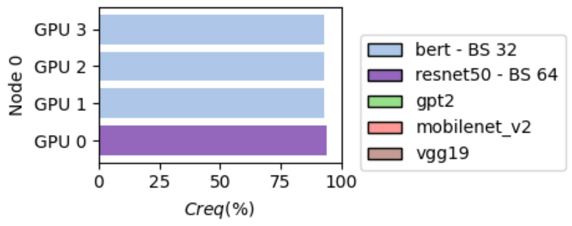


Figure 5.6: Model placement obtained from Usher for `alexnet`, `bert`, `gpt2`, `resnet50` and `vgg19` each having an expected request arrival rate of 400 requests per seconds and an SLO of 300ms. The x-axis shows the achieved occupancy.

2. To serve the entire load, `resnet50` and `alexnet` each require at least one replica, while `gpt2` and `t5` require at least 4 and 3 replicas of their highest batch sizes, respectively. This means the replication factors for `gpt2` are multiples of 4, and the replication factors for `t5` are multiples of 3.

3. Usher next generates the set of all possible configurations. Each configuration holds a `(rep_factor, batch_size)` tuple for each model.

4. Since all models are compute heavy (i.e $\frac{Creq}{Mreq} \geq 1.2$), Usher places the models in reverse order according to their total resource requirement $Rreq = Mreq + Creq$. However, there is no order in which `alexnet` has a higher $Rreq$ than `t5`, so it is never possible to have an order that places both `alexnet` and `resnet50` before `t5`.

5. As the $Creq$ of all models is greater than 69%, no model colocation occurs, and each GPU hosts exactly one replica.

6. The replication factor of 3 for `t5` means it will always occupy 3 out of the 4 GPUs if possible, leaving one GPU for an additional replica.

This example highlights how the combination of replication factors and placement order can lead to suboptimal outcomes in Usher's heuristic approach. Note also that despite having 4 GPUs, 2 out of 4 models are not being served at all.

Undefined Order across Groups

Having examined how the heuristic order within a group can lead to suboptimal outcomes and potentially prevent some models from being served, we now turn our attention to a similar issue that can arise due to the order in which groups of models are placed onto GPUs. As discussed in Section

Model	Batch Size	Latency(s)	Goodput(rps)	$Mreq(\%)$ [42]	$Creq(\%)$ [21]
alexnet	4	0.0014	2801.75	1.66	69.17
	8	0.0023	3540.12	2.08	85.97
	16	0.0031	5196.69	2.08	85.93
	32	0.0053	5990.46	2.80	89.94
	64	0.0097	6627.59	3.44	91.95
	128	0.0182	7023.69	6.40	93.02
bert	4	0.0341	117.34	3.47	85.89
	8	0.0658	121.53	4.06	88.46
	16	0.1281	124.88	5.25	91.29
	32	0.2439	131.19	7.63	92.90
	64	0.4853	131.88	12.40	93.83
	128	0.9647	132.68	21.92	94.21
gpt2	4	0.0369	108.28	5.80	91.28
	8	0.0732	109.23	10.48	93.05
	16	0.1435	111.49	19.85	93.84
	32	0.2730	117.21	38.58	94.27
resnet50	4	0.0068	589.78	1.16	87.39
	8	0.0096	829.08	1.77	90.83
	16	0.0160	998.99	2.70	92.63
	32	0.0300	1067.13	4.52	93.58
	64	0.0573	1117.12	8.16	93.95
	128	0.1113	1149.98	15.45	100.00
t5	4	0.0311	128.74	3.17	97.18
	8	0.0580	137.83	4.88	97.49
	16	0.1096	146.02	8.15	97.74
	32	0.2131	150.19	16.08	97.79
	64	0.4211	151.97	29.17	97.89

Table 5.2: Profiled latency, maximum reserved memory capacity and achieved occupancy for one iteration on a NVIDIA V100

5.2.2, Usher first clusters models into nearly equal-sized groups before placing them sequentially onto the GPUs. However, neither the paper nor the code specifies the order in which these groups should be placed. The issue we present here is independent of whether our implementation of the clustering algorithm is correct.

Consider a fixed-size cluster with 4 GPUs. We have three vision models `resnet50`, `vgg19`, `mobilenet_v2` and two language models `gpt2` and `bert` to be served. Each model has an expected request arrival rate of 400 requests per second (req/s) and a Service Level Objective (SLO) of 300 ms. Following the paper's specifications, we assume a maximum group size of 4. Figure 5.6 illustrates the resulting placement.

Based on our profiling measures, the clustering algorithm returns two groups: the first consisting of `bert`, `resnet50`, `gpt2` and the second group consisting of `vgg19`, `mobilenet_v2`. Another iteration of the clustering algorithm would have led to a group size of 5, which is not allowed given the maximum group size of 4. It is unclear which group should be placed

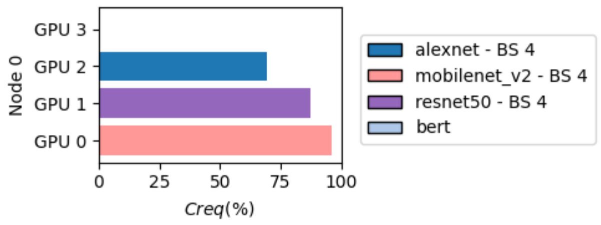


Figure 5.7: Model placement from Usher for `alexnet`, `resnet50`, `mobilenet_v2` and `bert` with an expected arrival rate of 500 req/s and an SLO of 200ms. The x-axis shows the achieved occupancy.

first, thereby having access to all the GPUs. Our implementation begins by placing the group `bert`, `resnet50`, `gpt2` onto the GPUs first.

Conducting a similar analysis as in the previous section, using the values from Table 5.2, one can verify that both `bert` and `gpt2` require a replication factor of 4 to meet their load. Given that the entire load of `resnet50` can be served with a single replica and no colocation takes place, the remaining 3 GPUs will be occupied by `bert` (which leads to a higher expected goodput than `gpt2`). As a consequence, all GPUs are occupied before the second group can even be placed.

However, the reader may convince themselves that any placement in which the group `vgg19`, `mobilenet_v2` would have been placed first would have resulted in a higher expected goodput.

This example highlights the significance of the order in which groups are placed across GPUs and how it can impact the overall system goodput. This example also shows that there is no notion of fairness between the models in the case of a fixed size cluster.

5.4.2 Problematic Replication Factor Multiplier

In Section 5.2.2, we discussed the ambiguities surrounding the replication factor of a model. To recap, the model's replication factor is chosen from the set $\{c_m^l \cdot i \mid c_m^l \cdot i \leq num\_gpus, i \in \{1, ..., 6\}\}$, where c_m^l is the minimum number of GPUs needed to serve a model's entire load within the SLO using the highest possible batch size.

Consider once again a fixed-size cluster with 4 GPUs. We have three vision models `resnet50`, `alexnet`, `mobilenet_v2` and `bert` as our only language model to be served. All models have an expected request arrival rate of 500 requests per second (different from the previous settings) and an SLO of 200 ms. The resulting model placement is shown in Figure 5.7.

Notably, `bert` is not being served, even though GPU 3 remains idle. From Table 5.2, we can see that the highest batch size that `bert` can run without violating the SLO is 16. This batch size results in an expected goodput of 124.88 req/s. Consequently, 5 replicas with batch size 16 would be required to satisfy the entire load of `bert` within the SLO. Given that our fixed-size cluster has only 4 GPUs, the set of replication factors to choose from for `bert` is empty.

According to our understanding of the Usher algorithm, in such a case, it determines that it will not be able to serve the workload of `bert` under any circumstances and therefore decides not to serve it at all. The result is that some GPUs might be left unused even though there is work to be scheduled on them. Note that if the expected goodput was only 0.12 req/s higher, Usher would have chosen to server `bert` as 4 replicas would have been sufficient.

In section 5.4.3, we propose to eliminate the replication factor multiplier c_m^l. This modification should prevent us from encountering this problem and allow for more flexible and efficient resource allocation, particularly in scenarios where partial serving of a model's workload is preferable to not serving it at all.

5.4.3 An Mixed Integer Linear Programming (MILP) Approach

To address the issues discussed in sections 5.4.1 and 5.4.2 regarding the placement order of models and the replication multiplier, we propose a Mixed Integer Linear Programming (MILP) approach based on the following modifications:

1. We eliminate group ordering and instead place all models together, resolving issues related to group placement priority.
2. We remove the heuristic order of placing models sequentially. Our MILP-based optimizer will return the global optimum in expected goodput, independent of any underlying heuristic order.
3. We remove the replication factor multiplier c_m^l, making it preferable to serve a model with one replica (even if not serving the entire workload) rather than not serving the model at all. This ensures that no GPUs are left unused as long as there is work to be scheduled on them.

Problem Definition

Table 5.3 introduces the notation for our solver. Note that the w_m variable is of type float, making this a mixed integer programming problem. Our solver aims to maximize the overall expected cluster goodput, similar to Usher. The goodput that a single model may contribute to this overall cluster goodput cannot exceed the model's actual expected arrival request rate. Thus, the objective of our solver can be expressed as:

Notation	**Definition**	**Variable Type**
M	The set of all models to serve	
B_m	The set of all possible batch sizes for a model m	
G	The set of all GPUs in the cluster	
max_rep_m	The maximum replication factor of a model m	
$goodput_{m,b}$	Expected goodput of model m with batch size b	`float`
rps_m	Expected request arrival rate for model m	`float`
$mreq_{m,b}$	Memory requirement of model m with batch size b	`float`
$creq_{m,b}$	Compute requirement of model m with batch size b	`float`
$x_{m,b,g}$	Model m runs with batch size b on GPU g	`binary`
$y_{m,b}$	Model m runs with batch size b	`binary`
w_m	Auxiliary variable to linearize min function for model m	`float`

Table 5.3: Variable definition for solving the Usher placement with a Mixed Integer Linear Programming Solver considering a homogeneous GPU cluster.

$$max \sum_m min(rps_m, goodput_m)$$

This objective criterion is non-linear due to the min function. We therefore introduce an auxiliary variable w_m to linearize the objective:

$$max \sum_m w_m$$

The objective criteria is subject to the following constraints for all models m. The goodput a single model m can contribute cannot be higher than its expected arrival rate.

$$w_m \leq rps_m$$
$$w_m \leq \sum_{b \in B_m} goodput_{m,b}$$

The batch size for a model m should be the same across all model replicas

$$\sum_{b \in B_m} y_{m,b} \leq 1$$

If a model m is placed on any GPU g with batch size b, the corresponding batch size b must be activated for model m

$$x_{m,b,g} \leq y_{m,b}$$

For each GPU g, the sum of $Creq$ and $Mreq$ of all models m and corresponding batch size b cannot exceed 100%

$$\sum_{m \in M, b \in B_m} creq_{m,b} \cdot x_{m,b,g} \leq 100$$
$$\sum_{m \in M, b \in B_m} mreq_{m,b} \cdot x_{m,b,g} \leq 100$$

For each GPU g and each model m, the model m may be placed at most once on GPU g

$$\sum_{b \in M_b} x_{m,b,g} \leq 1$$

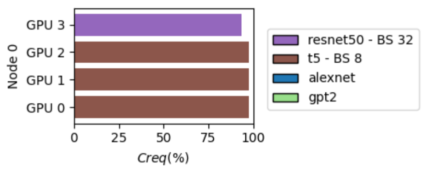
Each model m must not exceed its maximum replication factor

$$\sum_{m \in M, b \in M_b, g \in G} x_{m,b,g} \leq max\_rep_m$$

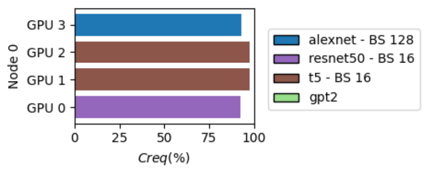
The above defined MILP problem could be extended to include a heterogeneous GPU cluster setting. In that case, the variables $goodput_{m,b}$, $creq_{m,b}$, and $mreq_{m,b}$ would need to be extended by a third GPU dimension. Note that this problem definition is agnostic to which concrete metrics we use for the $Creq$ or $Mreq$. This means that we can combine the MILP solver with any of the metrics that we have introduced in Section 5.3 and beyond.

We have implemented our MILP solver in Python based on the PuLP solver [36].

Revisiting the suboptimal Usher placements



(a) Usher

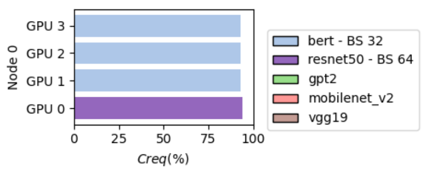


(b) MILP solver

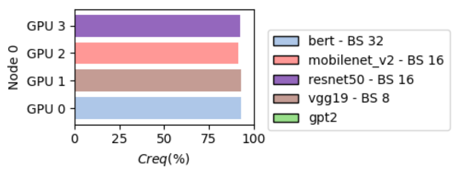
Figure 5.8: Model placement for `alexnet`, `resnet50`, `mobilenet_v2` and `bert` with an expected arrival rate of 500 req/s and an SLO of 200ms. The x-axis shows the achieved occupancy.

Figure 5.8 revisits the model placement from section 5.4.1 which was suboptimal due to the heuristic placement order within a group. The setting remains unchanged, and both policies use the achieved occupancy as $Creq$. In Figure 5.8a we see the previously analyzed Usher placement, which results in an expected goodput of 800 req/s. Meanwhile, Figure 5.8b shows the model placement obtained using the MILP solver-based placement policy, yielding an expected goodput of 1092.04 req/s. Unlike Usher, the MILP solver does not rely on any fixed order and instead places model replicas to maximize the expected goodput. Note the different choice of batch sizes as well between both policies.

Figure 5.9 similarly revisits the model placement but focuses on the suboptimal placement order across groups, as described in section 5.4.1. Usher's group-wise placement, shown in Figure 5.9a leads to an expected



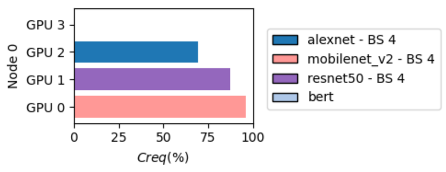
(a) Usher



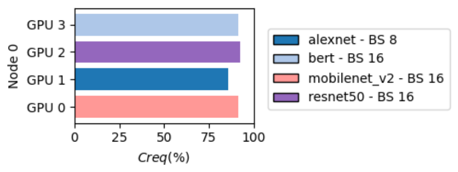
(b) MILP solver

Figure 5.9: Model placement for `alexnet`, `bert`, `gpt2`, `resnet50` and `vgg19` each having an expected request arrival rate of 400 requests per seconds and an SLO of 300ms. The x-axis shows the achieved occupancy.

goodput of 793.57 req/s. In contrast, Figure 5.9b depicts the MILP solver-based placement, which results in a substantially higher goodput of 1331.19 req/s. Unlike Usher, the MILP solver places all models at once rather than in groups.



(a) Usher



(b) MILP solver

Figure 5.10: Model placement for `alexnet`, `resnet50`, `mobilenet_v2` and `bert` with an expected arrival rate of 500 req/s and an SLO of 200ms. The x-axis shows the achieved occupancy.

Finally, Figure 5.10 revisits the placement issue related to the replication factor multiplier for `bert`, covered in section 5.4.2. In the Usher placement shown in Figure 5.10a one GPU remains idle due to the multiplier preventing `bert` from being scheduled. The MILP solver-based placement in Figure 5.10b however, uses all available GPUs, ignoring the replication factor multiplier and ensuring that resources are fully utilized wherever possible.

Downsides of the MILP

While our proposed Mixed Integer Linear Programming (MILP) approach addresses several challenges identified in the Usher algorithm, it is not without its own limitations. The significance of these limitations may vary depending on the specific criteria prioritized by users.

One notable drawback of our MILP solver is its lack of inherent fairness considerations. The solver may favor certain models over others based on their potential contribution to overall goodput. Models with higher expected arrival rates or those that achieve high expected goodput for their batch sizes will be prioritized when the cluster cannot serve the entire load of all models. This bias could result in some models being completely excluded from service. This limitation raises broader questions about what guarantees a cluster should provide. We will not further discuss this topic at this point and briefly revisit it in chapter 7.

The solver also has no knowledge over which batch sizes and replication factors to favor. Several configurations might result in the same maximum expected goodput. The solver will return one of them. We will see in chapter 6 that this choice may not always be ideal.

Another significant concern is the computational complexity of the solver. Mixed Integer Programming optimization is known to be NP-complete, with complexity growing exponentially in relation to the number of constraints and decision variables. Consequently, these factors need to be kept to a minimum for practical implementation. Our use of MILP in this context was primarily to evaluate the effects of removing heuristic orders and the replication factor multiplier. For the purposes of our evaluation, we consider this a reasonable approach, even though it may not be a scalable one for larger systems or more complex scenarios.

Chapter 6

Evaluation

This chapter presents a series of experiments that we have conducted in order to compare different placement policies to each other. Section 6.1 defines the evaluation environment as well as the different placement policies used for evaluation. Section 6.2 evaluates the effect on different placement policies when successively granting them more GPUs. We conclude this chapter by section 6.3, where we analyze the effect when increasing the load on the cluster for different Service Level Objectives.

6.1 Evaluation Environment

This section outlines the environment used to conduct the experiments in this study. It is organized as follows: Section 6.1.1 describes the machine configuration, Section 6.1.2 introduces the models used in the evaluation, and Section 6.1.3 lists the placement policies assessed during the experiments. Section 6.1.4 introduces the goodput metric which serves as the primary evaluation criterion, and finally, Section 6.1.5 deatils the client behavior.

6.1.1 Machine Setup

The experimental environment for this study is detailed in Table 6.1. As previously discussed in Section 5.1.4, we encountered compatibility issues between the NVIDIA Nsight Compute profiler and the NVIDIA Multi-Process Service for the profiling process of iGniter. This necessitated the use of an older NVIDIA GPU driver version, which subsequently required us to employ earlier versions of the operating system, profilers, and CUDA to ensure backward compatibility. It is important to note that this modified setup was exclusively applied to the iGniter nodes, while the node VMs of all other policies and the cluster controller VMs adhered to the general setup.

All experiments were conducted within the Google Cloud Platform (GCP)

	General Setup	iGniter
OS	Ubuntu 22.04	Ubuntu 20.04
Machine Type [10]	n1-standard-16	n1-standard-16
Number vCPUs	16	16
Number of Cores	8	8
Memory	60GB	60GB
CPU Platform	Intel Syklake	Intel Skylake
NVIDIA GPU Driver	550.54.15	470.256.02
NVIDIA Nsight Systems [28]	2023.4.4.54	2022.4.2.1
NVIDIA Nsight Compute [26]	2024.1.0.0	2022.3.0.0
GPU Type	Tesla V100-SXM2-16GB	Tesla V100-SXM2-16GB
Python	3.10.12	3.10.15
Torch	2.4	2.4
Torchvision	0.19.0	0.19.0
CUDA	12.4.0	11.8.0
cuDNN	9.1.0	9.1.0

Table 6.1: Evaluation Machine Setup

environment. We utilized separate VMs for the nodes and the cluster-level components. The VM hosting the cluster-level components was configured without any GPUs attached. For network consistency and performance, all VMs were integrated into a Virtual Private Cloud (VPC) [11] within the same Availability Zone (`us-central1-c`). To quantify the network performance within our experimental setup, we conducted measurements using `iperf3` [18] between two distinct VMs in the `us-central1-c` zone, both residing within the VPC. These tests revealed an average network bandwidth of 16.6 Gbps. Additionally, we measured the network latency between these VMs, observing an average ping latency of 0.322 ms.

6.1.2 Models and Input Data used for Evaluation

This section specifies the models and corresponding input data employed in our evaluation process. We categorize our models into two primary groups: vision models for object detection and language models for sentence classification.

Vision Models for Object Detection

Table 6.2 presents an complete list of the vision models utilized in this study. These models are loaded from `torchvision` [50] without any pre-trained weights. For each model, we conduct object detection across 1000 classes using randomly generated input data with dimensions $3 \times 224 \times 224$. As part of serving the client requests, we apply an `argmax` function to the probability tensor, returning only the predicted class label to the client.

Model	Num Params
Alexnet [41]	61.1M
DenseNet121 [43]	8.0M
EfficientNet-B7 [44]	66.3M
Inception-V3 [45]	27.2M
MobileNet-V2 [46]	3.5M
ResNet50 [48]	25.6M
VGG19 [49]	143.7M

Table 6.2: Vision Models used for Object Detection

Language Models for Sentence Classification

Model	Num Params	Weights	Vocab Size
BERT [2]	110M	`bert-base-uncased`	30522
BLOOM-560 [3]	560M	`bigscience/bloom-560m`	250680
GPT2 [15]	1.5B	`gpt2`	50257
T5 [52]	50M	`t5-small`	32000
XLNet [57]	110M	`xlnet-base-cased`	32000

Table 6.3: Language Models used for Sequence Classification

The language models employed in our evaluation are listed in Table 6.3. These models are imported from HuggingFace Transformers [16]. The table provides both the pre-trained weights utilized and the vocabulary size for each model. Our evaluation process involves performing sequence classification on input sequences of 512 tokens, with the input data randomly generated from the model's vocabulary. Similar to the object detection task, we apply an `argmax` function to the probability tensor and return only the predicted label to the client.

6.1.3 Placement Policies used for Evaluations

Several placement policies have been used as part of our evaluation. Each of them is defined below:

- **iGniter**: This policy represents our original implementation of the iGniter placement strategy, incorporating our updated profiled model and hardware coefficients. The worker nodes in this implementation use a worker that utilizes multiple streams to overlap data copying and computation of independent requests, while also setting the MPS share to limit the set of Streaming Multiprocessors available to the dispatcher.

- **Usher**: This is the original Usher placement policy based on our understanding of the paper. It utilizes the maximum achieved occupancy for the lifetime of the model as the $Creq$ metric and `torch.cuda.max_memory_reserved` as the $Mreq$ metric. The worker nodes in this implementation also employ streaming and set the MPS share. As stated in the Usher paper we use a maximum group size of 4 models per group in our clustering algorithm.
- **WAO-Usher** This policy, is a variant of the Usher implementation that employs weighted average achieved occupancy (Section 5.3.1) as the $Creq$ metric. It maintains the use of `torch.cuda.max_memory_reserved` as the $Mreq$ metric and implements both streaming and MPS share configuration at the worker level. It also uses a desired group size of 4 models per group in the clustering algorithm.
- **AO-MILP** This policy utilizes a Mixed Integer Linear Programming Solver (Section 5.4.3) for model placement on GPUs. It employs achieved occupancy as $Creq$ and `torch.cuda.max_memory_reserved` as $Mreq$. While it implements streaming at the worker level, it does not configure MPS share, allowing multiple workers to execute concurrently on the same GPU while sharing the same set of Streaming Multiprocessors.
- **WSM-MILP** This policy shares the underlying implementation of AO-MILP but utilizes weighted SM Utilization (Section 5.3.2) as the $Creq$ metric. It maintains the use of `torch.cuda.max_memory_reserved` as the $Mreq$ metric and implements streaming at the worker level without setting the MPS share.

6.1.4 Goodput as Evaluation Metric

To assess the efficacy of the various placement policies, we employ **goodput** as our primary evaluation metric. Goodput is defined as the number of requests per second that a cluster can successfully process while adhering to a specified Service Level Objective (SLO). This metric is distinct from traditional throughput measurements in that it incorporates both the quantitative aspect of request processing and the qualitative dimension of service quality. For a request to be considered valid and contribute to the goodput, its end-to-end latency must not exceed the predetermined SLO. Consequently, the goodput of a system is always less than or equal to its throughput.

6.1.5 Clients

In our experimental setup, we employ a one-to-one mapping between clients and models. For each model under evaluation, a dedicated client is instantiated with the following parameters:

1. Total number of requests to submit
2. Target submission rate (requests per second)
3. Service Level Objective (SLO) in milliseconds

The client operates by submitting request exclusively for its assigned model at the specified target rate until the total number of requests is exhausted. The client sender process operates asynchronously, submitting requests without waiting for the results of previous submissions. For a request to be considered valid, its end-to-end latency must be below the respective SLO.

To facilitate reproducibility across experimental runs, we implement deterministic request submission profiles by setting a fixed seed for each client. This approach allows us to maintain consistency in the request patterns, enabling more accurate comparisons between different placement policies and system configurations.

Prior to each experiment, the clients always submit a number of requests, not included in the total number of requests to be submitted, for a specified duration, in order to warm up the system.

6.2 Limiting the number of GPUs

This section presents our experimental findings on the performance of various placement policies as we incrementally increase the number of GPUs available to the cluster. We conduct two distinct experiments: Section 6.2.1 focuses exclusively on vision models, while Section 6.2.2 examines a heterogeneous workload comprising both vision and language models.

6.2.1 Limiting Number of GPUs with Vision Models

Experimental Setup

For this experiment, we utilize five vision models: `alexnet`, `densenet121`, `efficientnet_b7`, `resnet50`, and `vgg19`. Each model is subjected to a workload of 4000 requests with a target submission rate of 500 requests per second. We establish a Service Level Objective (SLO) of 200ms for all models.

Initially, we determined that the Usher policy requires six GPUs to fully serve this workload within the specified SLOs. Usher uses one replica for each model, except for `efficientnet_b7` it uses two as no batch size can serve the entire workload with a single replica. Figure 6.5a shows the corresponding placement. Given that Usher does not permit colocation when using achieved occupancy as *Creq*, this six-GPU configuration serves as an upper bound for our experiments. Our evaluation begins with a single-GPU cluster and incrementally adds one GPU per iteration, up to a maximum of six GPUs. This approach allows us to analyze the potential for achieving comparable goodput to the six-GPU Usher configuration with fewer GPUs, thereby exploring the trade-off between cluster performance and resource cost.

We employ two Virtual Machines (VMs) for this experiment. The first VM hosts all cluster-level components, while the second VM operates a node with eight NVIDIA V100 GPUs attached. We use the `CUDA_VISIBLE_DEVICES` variable in order to limit the number of GPUs that can be used in each iteration. Both VMs adhere to the configuration detailed in Section 6.1.1, with the exception of the iGniter setup, which utilizes a distinct configuration for the node-running VM.

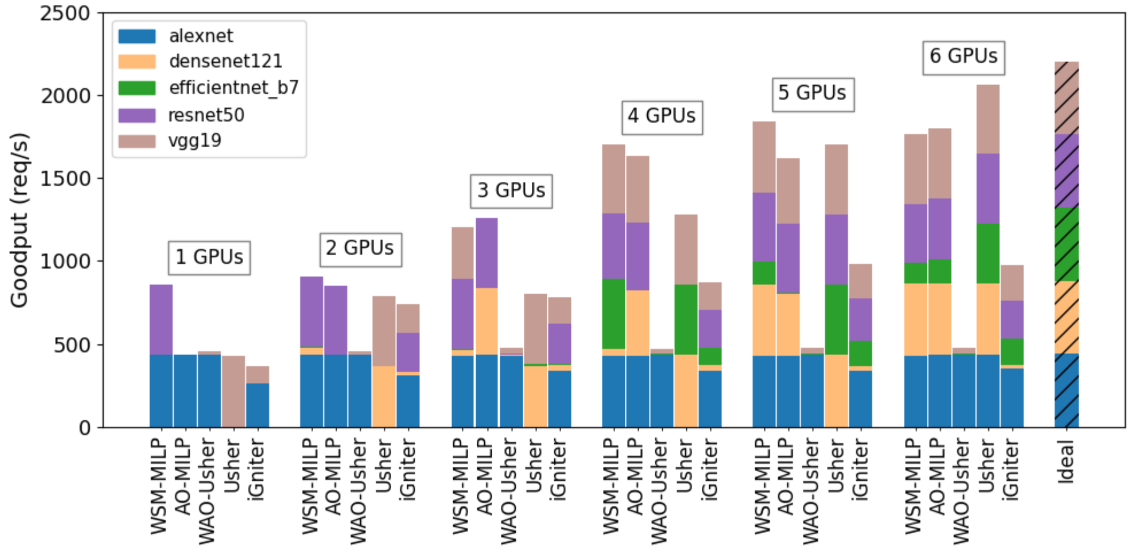


Figure 6.1: Achieved Goodput in req/s for five vision models on a cluster of up to six V100 GPUs

Results and Analysis

Figure 6.1 illustrates the experimental results, showing the achieved goodput in requests per second for each policy across varying numbers of GPUs. Each bar is segmented to show the individual contributions of each model to the overall goodput. The rightmost stacked bar represents the ideal goodput achievable if all requests are processed within their respective SLOs. Its segments correspond to each model's achieved target submission rate. We will first provide an analysis on the high-level trends of this experiment and focus on a series of specific scenarios after that.

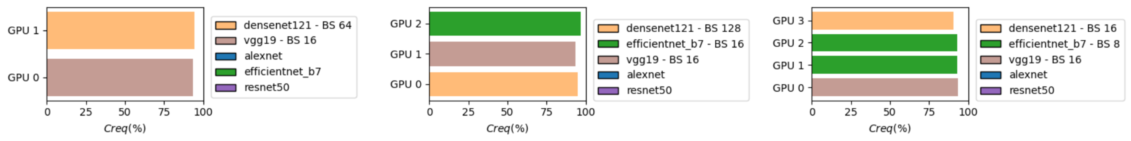
1. Usher demonstrates the best performance when allocated all six GPUs, achieving the highest goodput among all evaluated policies and approaching the ideal scenario.
2. WAO-Usher shows no significant improvement with additional GPUs, suggesting that weighted achieved occupancy is not an effective metric for $Creq$.
3. AO-MILP outperforms Usher, particularly for cluster sizes of three and four GPUs, indicating that the MILP solver can effectively circumvent suboptimal placements resulting from Usher's heuristic ordering.
4. WSM-MILP successfully colocates `alexnet` and `resnet50` on a single GPU. Moreover, it achieves goodput levels with four and five GPUs that are competitive to Usher's performance with six GPUs, presenting a cost-effective alternative for users willing to accept a marginal decrease in cluster performance.
5. iGniter demonstrates minimal improvement with each additional GPU, processing small quantities of requests for each model but maintaining overall poor performance.

We will next take a deeper dive into some very specific scenarios. To not overload this report, we have compiled a representative list of these scenarios meant to identify further shortcomings of the different policies as well as decisions that are crucial to the cluster's performance. We refer the reader to Table A.1 in the Appendix that contains crucial metrics that will be used throughout the whole analysis. For space reasons we have decided to put the table into the Appendix.

Observation 1: Usher's goodput increase from two to four GPUs

Observation Usher's goodput only marginally increases when going from two to three GPUs, but increases significantly when going from three to four GPUs. Figure 6.2 shows Usher's placement for two, three and four GPUs. With each additional GPU that is added to the cluster, Usher can place one more replica on the new GPU. Its goodput however only marginally improves when going from two to three GPUs despite the addition of the `efficientnet_b7` replica on GPU 2. According to our profiling results (Table A.1 in the Appendix), `efficientnet_b7` has an expected goodput of 344.1 req/s for batch size 16. This is insufficient to meet the target submission rate of 500 req/s, resulting in queue buildup in the worker's input buffer due to the mismatch between incoming request rate and processing capacity. The situation improves with four GPUs. Usher now deploys two replicas of `efficientnet_b7`, each with a batch size of 8 and an expected goodput of 260.14 req/s per replica. With each replica handling an expected arrival rate of 250 req/s, queuing delays are reduced, and most requests can be processed within their specified SLO.

Key Takeaway Having multiple replicas of a model (possibly with smaller batch sizes) can help reduce queuing delays, while keeping latency within the SLO.



(a) 2 GPUs (b) 3 GPUs (c) 4 GPUs

Figure 6.2: Usher Model Placement

Observation 2: WAO-Usher performs poorly compared to other baselines

Observation WAO-Usher's overall goodput remains stagnant despite the addition of GPUs to the cluster. Figure 6.3 shows the model placement for WAO-Usher with four GPUs. We observe that only three out of four GPUs are used. WAO-Usher employs this same placement policy for any cluster of more than three GPUs.

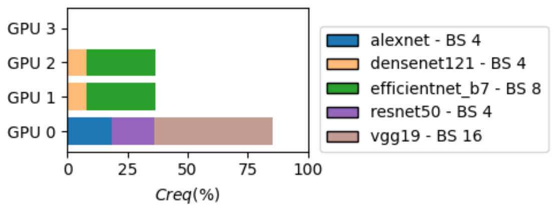


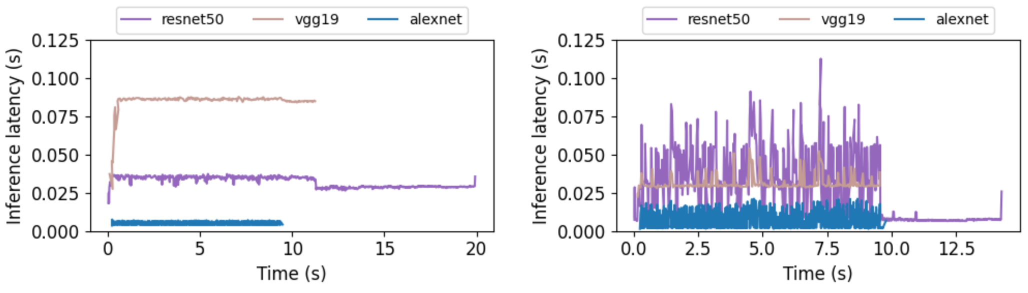
Figure 6.3: Model Placement for WAO-Usher for four GPUs

An examination of inference latencies on GPU 0 for the entire time of the experiment (Figure 6.4a) reveals that `resnet50` with batch size 4 suffers from an approximate 4x increase in inference latency compared to its profiled inference latency of 6.8ms when run in isolation with access to all SMs on the GPU (Table A.1). The fact that the inference latency of `resnet50` only slightly decreases after `vgg19` and `alexnet` have terminated, suggests that the main reason for `resnet50`'s high inference latency is due to it being constrained only to a small set of SMs on the GPU given its small *Creq* share of 17.55%.

As a result, we conducted the exact same experiment, this time without setting MPS shares for both Usher and WAO-Usher. The overall results of this experiment are presented in Figure A.1 of the Appendix. Figure 6.4b illustrates the inference latency on GPU 0 for this modified setup.

Our findings indicate that while the models are no longer constrained to specific SM sets, `resnet50` and `alexnet` experience significant interference. The inference latency of `resnet50` decreases substantially after all other models have completed execution, allowing it to utilize all available SMs on the GPU. Although not setting MPS shares for WAO-Usher resulted in a slight improvements in overall goodput, its aggressive colocation strategy led to severe inter-model interference. Consequently, the overall goodput of WAO-Usher remains sub optimal compared to other evaluated policies.

Key Takeaway Using the weighted achieved occupancy as a metric leads to suboptimal decisions that greatly harm models' latency and throughput. This is due to underestimating the amount of resources the workloads need, leading either to constraining models to only a few SMs, or high interference upon free SM sharing



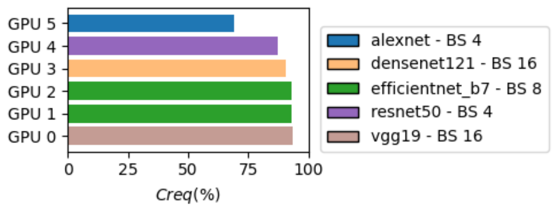
(a) setting MPS share (b) not setting MPS share

Figure 6.4: Inference latency for WAO-Usher on GPU 0

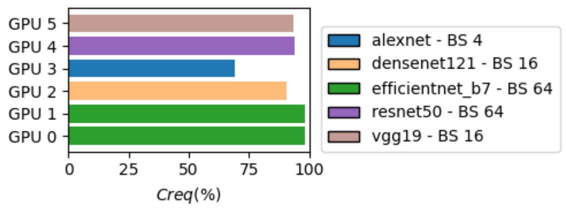
Observation 3: AO-MILP performs worse than Usher on six GPUs

Observation Despite the introduction of AO-MILP as an enhancement over Usher to mitigate suboptimal outcomes resulting from heuristic model placements, our experimental results reveal that AO-MILP performs worse than Usher on six GPUs. Both policies employ the achieved occupancy (AO) as the *Creq* metric, yet AO-MILP's overall goodput is inferior to Usher's, particularly for `efficientnet_b7`.

Model Placement Comparison Figure 6.5 illustrates the divergence in model placement strategies between the two policies. The main difference lies in the batch sizes selected for `efficientnet_b7` (8 for Usher vs. 64 for AO-MILP) and `resnet50` (4 for Usher vs. 64 for AO-MILP).

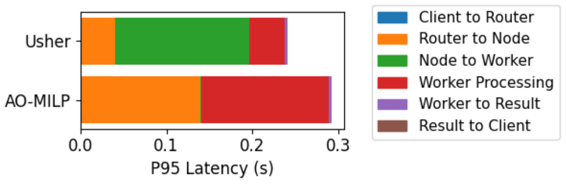


(a) Usher

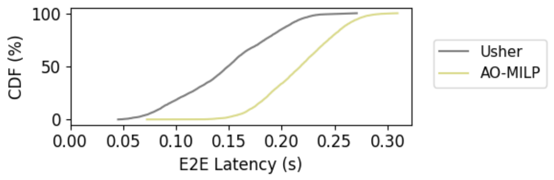


(b) AO-MILP

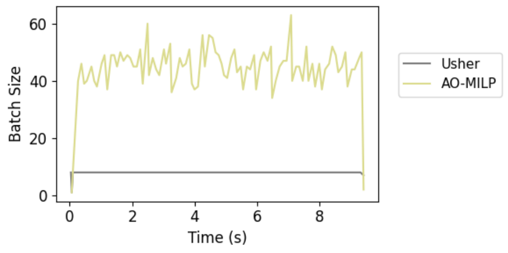
Figure 6.5: Model Placement on six GPUs



(a) Latency Breakdown



(b) E2E latency CDF



(c) Batch Size

Figure 6.6: Comparing AO-ILP and Usher for `efficientnet_b7`

Latency Breakdown Analysis To analyze where the performance difference comes from, we conducted a detailed latency breakdown analysis, as depicted in Figure 6.6a. This visualization breaks the end-to-end latency of each request into distinct segments, presenting the 95th percentile latency for each segment. The most important segments are defined below:

- **Router to Node** is the time from request arriving at the Router (Section 4.3.3) until the reception at the Node Controller Server (Section 4.4.1). This includes the batch build time as well as the network transmission latency from the router to the node.
- **Node to Worker** is the time spent in a worker's input queue awaiting GPU processing
- **Worker Processing** is the time for request processing by the worker. This time includes GPU inference but excludes result transmission to

the Result Server (Section 4.3.4).

The plot reveals that Usher's requests mainly experience delays in the worker's queue, indicating minor queue buildup at the worker level. On the other hand, AO-MILP's requests exhibit longer Router to Node transition times and extended Worker Processing durations, with negligible worker queue delays.

Batch Size Impact The performance divergence can be attributed to the separate batch sizes used by each policy for `efficientnet_b7`. AO-MILP's larger batch size (64 vs. 8) results in extended batch formation and transmission times. Figure 6.6c demonstrates that AO-MILP frequently fails to achieve its target batch size of 64, instead dispatching batches due to the `MAX_WAIT_TIMEOUT` (set at 100ms).

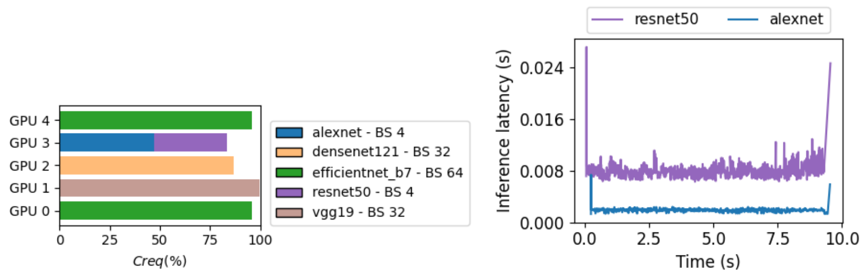
Moreover, larger batch sizes result in higher GPU processing times. From Table A.1, we observe that `efficientnet_b7` inference latency increases from 0.0308s for batch size 8 to 0.1609s for batch size 64. The combination of prolonged batch formation and inference times renders AO-MILP incapable of meeting the 200ms SLO under these conditions.

Throughput vs. Goodput Interestingly, both placements achieve nearly identical throughput for `efficientnet_b7` (425.75 req/s for Usher vs. 425.25 req/s for AO-MILP). However, their goodput metrics diverge significantly: 360.61 req/s for Usher compared to 143.73 req/s for AO-MILP. Figure 6.6b visualizes this discrepancy. It shows the cumulative distribution function (CDF) of end-to-end latencies for both policies. The CDF curve for AO-MILP is shifted more to the right, since individual requests take longer to complete. As a result fewer request meet the SLO.

Key takeaway Large batch sizes can be detrimental for goodput due to the extra time needed for batch formation, and transmission times. A system should accurately model these overheads to make accurate decisions.

Observation 4: WSM-MILP: Successful collocation despite interference

Observation The WSM-MILP scenario with five GPUs gets closest to the overall goodput achieved by Usher with six GPUs. This example shows how successful colocation can be achieved despite a modest level of interference. Figure 6.7a illustrates the model placement for WSM-MILP. Of particular interest is the colocation of `resnet50` (batch size 4) and `alexnet` (batch size 4) on GPU 3. It should be reminded that, unlike other policies, WSM-MILP does not implement MPS share settings at the worker level.



(a) Model Placement (b) Inference Latency on GPU 3

Figure 6.7: WSM-ILP on five GPUs

Figure 6.7b presents the inference latency over time for GPU 3. A comparison of the observed inference latency for `resnet50` (approximately 8ms) with its profiled inference latency for batch size 4 (6.8ms) reveals a slight interference effect. Despite this interference, the worker demonstrates the capability to process most of the workload within the specified SLO. With an inference latency of approximately 8ms, the processing rate of the worker for `resnet50` can be estimated at roughly 500 req/s. Given that the achieved submission rate for `resnet50`'s worker is 433.34 req/s, the worker's processing capacity exceeds the achieved arrival rate. A similar analysis can be conducted for `alexnet`.

This example underlines that the ability to accurately model interference resulting from colocation is crucial for assessing a model's capacity to serve its workload effectively. Despite some level of interference, models can still meet their performance objectives if properly managed. Workers with processing rates significantly higher than their expected arrival rates may be able to tolerate a degree of interference while still meeting workload demands. Conversely, workers operating near their maximum processing capacity become highly susceptible to even minor interference effects.

Note that despite successful colocation, this model placement is not able to serve `efficientnet_b7` effectively due to the same reasons outlined in the previous scenario.

Key Takeaway The level of intereference each model can tolerate varies for each model, depending on its base latency, clients' request rates, and SLOs. Using WSM as a model's *Creq* can model interference more accurately than WAO.

Observation 5: iGniter performs poorly when using more than three GPUs

Observation iGniter's overall goodput is fairly poor compared to policies such as WSM-MILP or AO-MILP. With the exception of `densenet121`, all models contribute to the total goodput, yet they all fail of reaching their individual ideal performance.

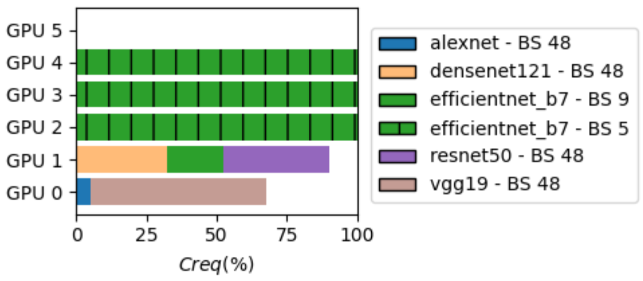
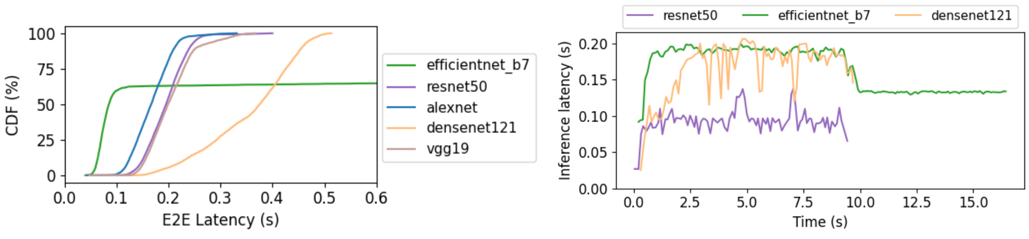


Figure 6.8: Model Placement for iGniter on six GPUs.

Figure 6.8 illustrates iGniter's model placement on six GPUs. According to the iGniter policy, five GPUs are required to serve the entire workload within the specified SLO. It is worth recalling that we have transformed iGniter into a best-effort policy (Section 5.1.5). iGniter first creates a model placement plan independent of the available GPU count in the cluster. Once the placement is determined, it is mapped onto the GPUs until either all GPUs are utilized or all replicas are placed. In this experiment, with one GPU, we begin by placing `vgg19` and `alexnet` on GPU 0. Given two GPUs, we can additionally place `densenet121`, `resnet50`, and a replica of `efficientnet_b7` on GPU 2, and so forth.

Unlike other policies examined thus far, iGniter may use different batch sizes across various model replicas, as is the case for `efficientnet_b7`. As specified in Section 4.3.3, the router distributes batches across model replicas in a round-robin fashion. Consequently, each replica receives a portion of the total workload relative to its batch size. In this case, the `efficientnet_b7` replica on GPU 1 receives $\frac{9}{24}$ of the load, while the other three replicas each receive $\frac{5}{24}$ of the entire load.

Figure 6.9a presents the end-to-end latency CDF for each model, truncated at 0.6s for readability. The CDFs for `alexnet`, `resnet50`, and `densenet121` demonstrate that no significant queuing occurs at their respective workers in the cluster. Nevertheless, a substantial proportion of requests still violate the 200ms SLO, once more primarily due to the large batch sizes chosen for these models. For `efficientnet_b7` we see how the load shares translate into the CDF. Approximately 62.5% of requests are served within 100ms, corresponding to the combined share of the three `efficientnet_b7` replicas



(a) End-to-end latency CDF

(b) Inference latencies on GPU 1

Figure 6.9: iGniter latencies on six GPUs

running in isolation on GPUs 2, 3, and 4. The remaining 37.5% correspond to the requests served on GPU 1, which experience substantial amounts of queuing at the worker level due to interference among the colocated models, as depicted in Figure 6.9b.

Key Takeaway iGniter fails to accurately model interference in all cases, resulting in large batch sizes which hurt goodput.

Key Insights and Implications

Here we generalize the key insights mentioned above:

1. The choice of batch sizes is crucial for inference. While larger batch sizes may enhance GPU utilization in training scenarios, they can be detrimental in inference contexts due to SLO constraints.

2. Load balancing the workload across multiple replicas with smaller batch sizes can help reduce queuing delays (compared to using large batch sizes).

3. Any placement policy that will estimate the goodput of a cluster without modelling intra-cluster queuing delays, will prove ineffective.

4. In order to be able to predict whether a worker is able to serve its workload, a placement policy must be able to model the interference between collocated models. Profiling models in isolation and summing up their individual results is insufficient to quantify whether there will be interference or not.

5. Cluster parameters like `MAX_WAIT_TIME` must be chosen with care.

6. A limitation of the MILP solver is its inability to differentiate between batch sizes that result in equivalent cluster goodput. Its repeated choice of batch size 64 for `efficientnet_b7` proved to be ineffective, while smaller batch sizes were possible.

We conducted the same set of experiments with a target submission rate of 1000 requests per second (req/s) for each model, maintaining the Service Level Objective (SLO) at 200 ms per model. In this scenario, the cluster size was scaled up to 8 GPUs. However, we have chosen not to provide a detailed analysis of these results, as they do not provide any significant new insights. The reader is invited to consult the corresponding plot in Figure A.2 of the Appendix for further information.

6.2.2 Limiting the Number of GPUs for Vision and Language Models

This experiment evaluates the different scheduling policies when language models are also considered. We used two vision models (`resnet50` and `mobilenet_v2`) and two language models (`bert` and `gpt2`). Each model processed 2400 requests at a target submission rate of 300 req/s, with a Service Level Objective (SLO) of 300ms for all models.

Profiling results revealed that serving language models is significantly more resource-intensive than vision models. Consequently, we adjusted the target submission rate and increased the SLO. To ensure fair comparison, we maintained consistent load across all models, preventing some models to be preferred over others based on goodput contribution. We excluded AO-MILP from this analysis as it does not colocate models.

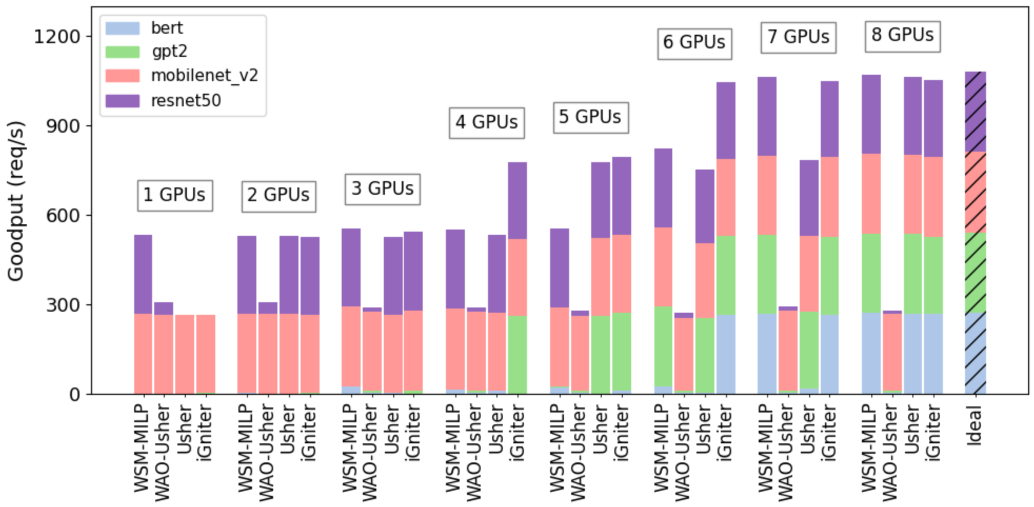


Figure 6.10: Achieved Goodput in req/s for language and vision models in a cluster of up to eight V100 GPUs

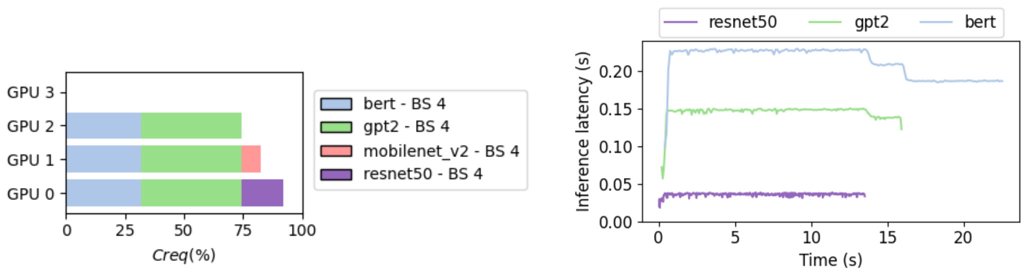
Figure 6.10 illustrates the results of the experiment. iGniter demonstrates strong performance, achieving near-ideal overall cluster goodput with just six GPUs. In contrast, WAO-Usher shows minimal improvement beyond a single GPU, maintaining poor performance. WSM-MILP presents competitive overall goodput in clusters exceeding five GPUs. Once again we have chosen a non-exhaustive list of specific scenarios that reveal key insights into cluster performance under different policies.

Observation 1: WAO-Usher performs poorly

Observation Figure 6.11a shows the model placement of WAO-Usher in a four-GPU cluster. WAO-Usher predicts three GPUs to be sufficient to satisfy the entire workload, colocating `bert` and `gpt2` on each GPU. Figure 6.11b presents the inference latency over time on GPU 0. As observed in the previous experiment, aggressive colocation leads to inter-model interference, that is nicely visible with `bert`'s latency, which improves as other models terminate. This example shows that despite the models being constrained to distinct sets of SM, the models still experience interference not expressed by the weighted achieved occupancy.

The primary cause of low goodput, however, is the MPS share limiting available SMs to each model. When `bert` runs alone on GPU 0 with batch size 4, its inference latency is approximately 180ms, resulting in an approximate processing rate of 22 req/s per replica. Clearly, three replicas of `bert` are insufficient to meet the expected submission rate of 300 req/s.

Key Takeaway WAO underestimates the models' resource requirements leading to aggressive colocation. Even when constraining the models to different SMs, there is still interference for shared resources such as L2 cache and GPU memory.

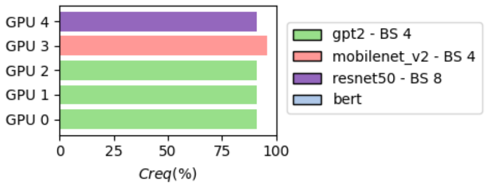


(a) Model Placement (b) Inference Latency on GPU 0

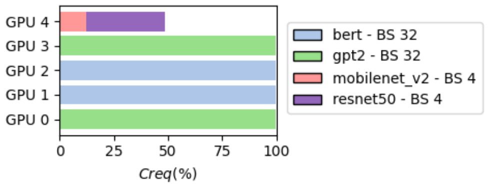
Figure 6.11: WAO-Usher on four GPUs

Observation 2: With five GPUs, WSM-ILP performs significantly worse than Usher

Observation At a cluster size of five GPUs, WSM-MILP demonstrates significantly inferior performance compared to Usher. Figure 6.12 illustrates the corresponding placements for both policies. Usher allocates three replicas of `gpt2` on GPUs 0 to 2. In contrast, WSM-MILP colocates `resnet50` and `mobilenet_v2` on GPU 4, utilizing the remaining four GPUs to place two replicas each of `bert` and `gpt2`. Neither `bert` nor `gpt2` make significant contributions to WSM-MILP's overall goodput.



(a) Usher



(b) WSM-MILP

Figure 6.12: Model Placements on five GPUs

Model	**Batch Size**	**Latency (s)**	**Estimated Goodput (req/s)**	**weighted SM Utilization (%)**
`bert`	4	0.0341	117.34	96.82
	8	0.0658	121.53	98.92
	16	0.1281	124.88	99.51
	32	0.2439	131.19	99.77
`gpt2`	4	0.0369	108.28	97.73
	8	0.0732	109.23	98.84
	16	0.1435	111.49	99.55
	32	0.2730	117.21	99.77

Table 6.4: Profiled metrics for `bert` and `gpt2` on a NVIDIA V100 GPU.

Table 6.4 summarizes the metrics to understand WSM-MILP's placement. The table includes only batch sizes with inference latency within the 300ms SLO. The **Estimated Goodput** column indicates that at least three replicas are necessary for both `bert` and `gpt2` to fully serve their load. Usher achieves this for `gpt2`, consequently serving most requests within the SLO. Conversely, WSM-MILP decides to place two replicas each of `bert` and `gpt2`, rather than fully serving one model and placing a single replica of the other. With two replicas for each model, neither can match the target submission rate of 300 req/s, resulting in queue buildup at their workers and most request for `bert` and `gpt2` violating their SLO.

WSM-MILP's placement is due to optimizing solely for goodput. Given the 300 req/s target submission rate for both models, placing two replicas of `bert` with batch size 32 yields an expected goodput of $2 \cdot 131.19 = 262.38$ req/s. Since a model cannot exceed its target submission rate in overall goodput contribution, a third replica of `bert` would only contribute $300 - 262.38 = 37.62$ req/s. From the MILP optimizer's point of view, it is more beneficial of placing a replica for `gpt2` with batch size 32, contributing 117.21 req/s to the total goodput. This scenario repeats for the second `gpt2` replica.

Figure 6.13 demonstrates how queuing at the Worker (**Node to Worker**) decreases for `bert` and `gpt2` as GPUs are added to the cluster. With six GPUs, WSM-MILP can add the third replica required for `gpt2`, while `bert` requests still violate the SLO. Finally, with seven GPUs, the third `bert` replica can be added, allowing WSM-MILP to achieve near-optimal goodput.

Key Takeaway Trying to maximize overall goodput might not always lead to optimal performance, especially when the system does not account for any queuing delays that might harm goodput, and lead to the actual goodput being significantly worse than the estimated.

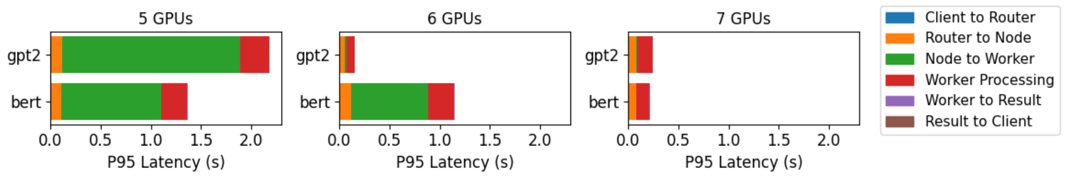
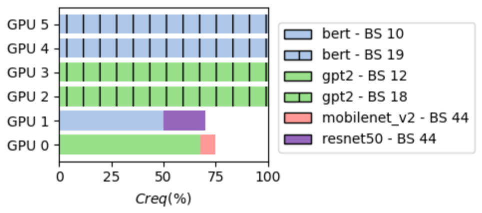


Figure 6.13: P95 latency breakdown for `bert` and `gpt2` in the WSM-MILP placement

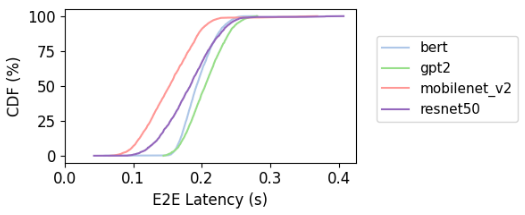
Observation 3: iGniter saves GPUs through efficient colocation

Observation With only six GPUs, iGniter demonstrates the capability to achieve near-optimal goodput across all models. Figure 6.14 illustrates both the model placement and end-to-end latencies for iGniter in a six-GPU cluster configuration. Notably, smaller batch sizes are selected for the `bert` and `gpt2` replicas colocated with `mobilenet_v2` and `resnet50`, respectively. The Cumulative Distribution Functions (CDFs) indicate that, despite colocation, requests for all models are smoothly served within their Service Level Objective (SLO). The increase in SLO from 200ms to 300ms mitigates the challenges for `mobilenet_v2` and `resnet50` that we have previously encountered with larger batch sizes in scenarios with a 200ms SLO.

Note that iGniter's goodput remains constant when increasing the cluster size from three to four GPUs. As previously mentioned, iGniter develops a placement plan designed to serve the entire workload, irrespective of a specific cluster size. Instead, it aims to minimize the number of GPUs required to serve the complete load. For instance, when evaluating iGniter with three GPUs, we apply its six-GPU placement policy and submit the entire load only to the first three GPUs. Consequently, for some models, the load exceeds the capacity of the replicas present in the first three GPUs.



(a) Model Placement



(b) End-to-end latencies

Figure 6.14: iGniter on six GPUs

Such is the case for `gpt2` with a cluster size of three. The first three GPUs only contain two replicas of `gpt2`, which are insufficient to handle a target submission rate of 300 req/s. As a result, most `gpt2` requests fail to meet their SLO. Only with the addition of the third `gpt2` replica does the cluster fully serve all requests within their SLO when adding a fourth GPU to the cluster. A similar pattern is observed for `bert` when transitioning from five to six GPUs.

Key Insights

The cases of iGniter and WSM-MILP have demonstrated that careful colocation of vision models, or vision models with language models, can effectively reduce the total number of required GPUs while maintaining high or even ideal overall goodput compared to placements that exclusively allocate individual GPUs to model replicas. However, the case of WAO-Usher has underscored the importance of carefully selecting colocation criteria, as overly aggressive colocation can prove detrimental to overall cluster goodput.

6.3 Changing the RPS and SLO

Our previous experiments have employed specific target submission rates and Service Level Objectives (SLOs). However, the appropriate selection of SLOs remains a complex issue, as they are inherently dependent on the underlying tasks they aim to address, and different clients may associate varying SLOs with identical tasks. For instance, Usher proposes setting a model's SLO at twice the average inference latency of a single request on an NVIDIA V100 GPU. They report an SLO of 108ms for `resnet50`, implying a single inference request time of 54ms. Our profiling, however, indicates an inference latency of 6.8ms for `resnet50` with a batch size of 4 on an NVIDIA V100 GPU, including data transfer. Following Usher's approach, we would set an SLO of approximately 13ms, which appears extremely challenging to meet given the necessity of routing requests through an entire cluster network.

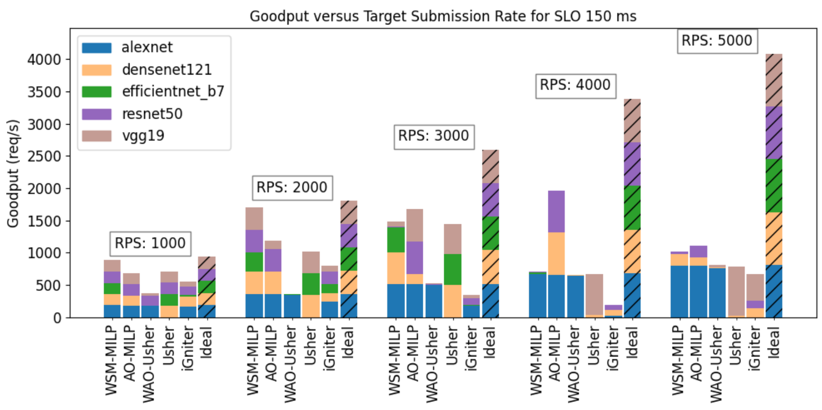
We have decided to conduct an experiment designed to evaluate the effect of different policies in a fixed-size cluster while incrementally increasing SLOs and target request submission rates. Our experimental setup consists of a 4-GPU cluster running five vision models: `alexnet`, `densenet121`, `efficientnet_b7`, `resnet50`, and `vgg19`. We consider SLOs of 150ms, 250ms, and 350ms. For each SLO, we increase the target submission rate from 1000 req/s to 5000 req/s in increments of 1000 req/s. The total load is equally distributed among all models.

Global Analysis and Insights

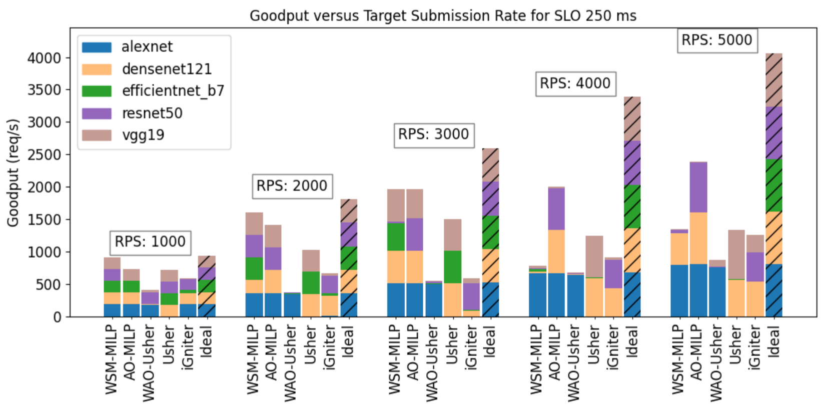
Figure 6.15 illustrates the results of these three experiments. For each fixed SLO, we plot the total goodput achieved by each policy as we increase the system load. Each policy's total achieved goodput is further subdivided into the goodputs contributed by individual models. The *ideal* bar for each target submission rate represents the maximum attainable goodput, corresponding to the achieved submission rates for each model.

The general trends across all three plots remain very consistent, and despite increasing the SLO, the goodput of different policies does not significantly increase across various loads. As the load intensifies, we observe a diminishing number of models being served and contributing to the overall goodput of a policy. We have identified several reasons for this phenomenon:

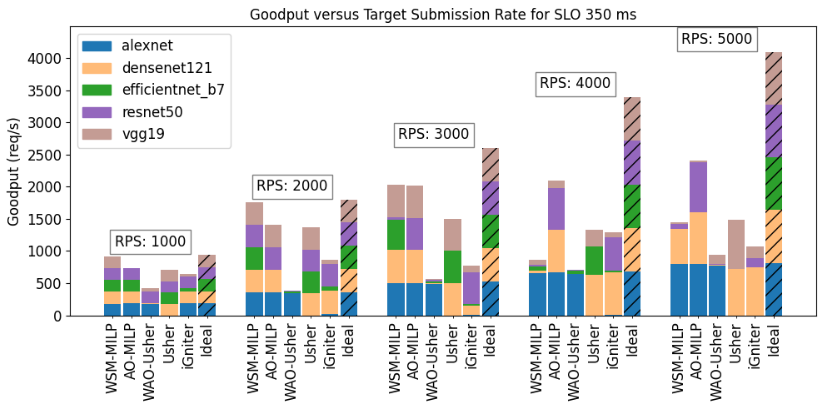
1. **Increasing resource demands**: As the load for a model increases, more GPU resources are required to serve the load. Since the evaluated policies tend to maximize the overall cluster goodput, they tend to favor models that are less resource-intensive to serve (e.g. `alexnet` or `densenet121`). Heavier models like `efficientnet_b7` tend to be not served at all or are not assigned enough processing capacity.



(a) SLO 150ms



(b) SLO 250ms



(c) SLO 350ms

Figure 6.15: Goodput versus Target Submission Rate in a cluster of 4 NVIDIA V100 GPUs.

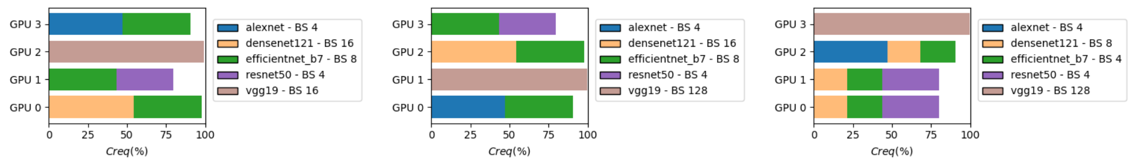
2. **Preference for larger batch sizes**: Larger batch sizes are often preferred over smaller ones as they typically present a higher expected goodput and are used as a means to meet increased load. However, we have observed that large batch sizes can be problematic for inference due to longer batch build times and increased inference latencies. While the system's throughput can be increased or at least maintained, the end-to-end latencies of individual requests increase.

3. **Worker-level queuing**: Once a model's load exceeds the processing capacity of its workers, queuing at the worker level builds up, becoming detrimental to all requests for that model. Whether the SLO is 150ms or 350ms does not make a large difference in that case because the overheads due to queuing are significantly larger.

Figure 6.15 also shows that none of the policies that colocate models perform well beyond 3000 RPS. For WAO-Usher, we have already seen that the weighted achieved occupancy is not the appropriate metric, and its aggressive and overly optimistic colocation leads to both interference and inefficient use of available resources. We'll take a closer look to specific cases of WSM-MILP, AO-MILP and iGniter below.

Observation 1: WSM-MILP Performance for SLO 250ms and Target Load exceeding 3000 req/s

Observation For the WSM-MILP policy, we observe a consistent pattern across all three SLOs: the total goodput decreases when the RPS goes from 3000 to 4000, then increases again when the RPS goes from 4000 to 5000. To understand this behavior, we will examine the model placements for target loads of 3000, 4000 and 5000 RPS for an SLO of 250ms, illustrated in Figure 6.16.



(a) RPS 3000 (b) RPS 4000 (c) RPS 5000

Figure 6.16: Model Placement for WSM-MILP for an SLO of 250ms

RPS 3000 vs RPS 4000 The placement policy for 3000 req/s and 4000 req/s differs solely in the batch size chosen for `vgg19` (16 vs 128). As the total target submission rate increases from 3000 to 4000 req/s, the target submission rate for each client rises from 600 req/s to 800 req/s. Consequently, it becomes impossible to serve `vgg19` with a single replica, as evidenced by the profiled values in Table 6.5. The MILP solver thus opts for a batch size of 128, which yields the highest expected goodput with a single replica. However, the inference latency of `vgg19` with this batch size is already 199ms. Furthermore, given that the processing capacity of the `vgg19` worker is at most 640 req/s, queuing will inevitably build up, causing most requests to violate the SLO in the RPS 4000 case.

Model	**Batch Size**	**Latency (s)**	**Estimated Goodput (req/s)**	**weighted SM Utilization(%)**
alexnet	4	0.0014	2801.75	47.07
	8	0.0023	3540.12	83.79
	16	0.0031	5196.69	63.49
	32	0.0053	5990.46	96.24
	64	0.0097	6627.59	91.56
	128	0.0182	7023.69	99.0
densenet121	4	0.0154	260.13	13.90
	8	0.0169	472.72	21.26
	16	0.0192	832.59	54.10
	32	0.0335	954.82	86.92
	64	0.0629	1017.07	92.30
	128	0.1203	1063.81	97.81
efficientnet_b7	4	0.0299	133.93	22.47
	8	0.0308	260.14	43.55
	16	0.0465	344.10	81.32
	32	0.0883	362.31	92.35
	64	0.1609	397.70	95.81
resnet50	4	0.0068	589.78	36.26
	8	0.0096	829.08	70.49
	16	0.0160	998.99	87.33
	32	0.0300	1067.13	93.90
	64	0.0573	1117.12	98.04
	128	0.1113	1149.98	99.16
vgg19	4	0.0098	408.51	95.18
	8	0.0182	438.65	98.88
	16	0.0262	610.11	99.27
	32	0.0516	620.69	99.57
	64	0.1021	627.01	99.69
	128	0.1994	641.97	99.76

Table 6.5: Profiled metrics for `alexnet`, `densenet121`, `efficientnet_b7`, `resnet50` and `vgg19` on a NVIDIA V100 GPU.

The placement for all other models remains unchanged when increasing the load from 3000 to 4000 req/s:

- `resnet50`: Even at a target submission rate of 600 req/s, `resnet50` could not be served, as its worker's processing capacity with batch size 4 is approximately 590 req/s. At 800 req/s, the situation only becomes worse.

- `densenet121`: With batch size 16, the `densenet121` worker has a processing capacity of about 830 req/s. At 600 req/s (RPS 3000), the worker has some buffer to accommodate moderate interference from the colocated `efficientnet_b7`. However, at 800 req/s (RPS 4000), this buffer is minimal, and even slight interference creates worker-level queuing. Figure 6.17 illustrates this behavior, showing that already at 600 req/s, the requests experience slight delays in the worker input queue. This delay drastically increases at 800 req/s. This scenario perfectly demonstrates that once queuing builds up, increasing the SLO by a few tens or hundreds of milliseconds becomes indifferent. Any policy that fails to account for interference and model the resulting queuing effects will become ineffective as the system load increases.

- `efficientnet_b7`: When the target submission rate increases from 600 req/s to 800 req/s, the combined processing capacity of the three `efficientnet_b7` replicas can no longer meet the workload. Using a higher batch size would roughly double its weighted SM utilization. Given the cluster's 4-GPU limit, this would require sacrificing other models, which is suboptimal for overall goodput.

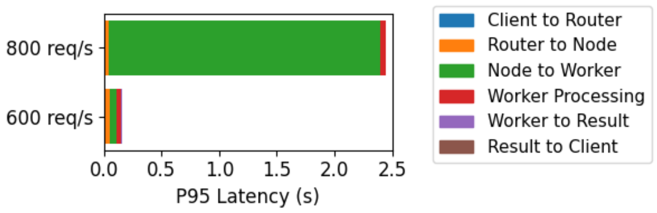


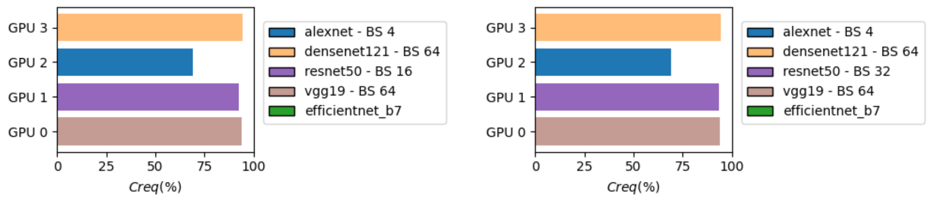
Figure 6.17: P95 latency breakdown for `densenet121` at a target submission load of 600 req/s and 800 req/s

RPS 4000 vs RPS 5000 As the target submission rate for each client further increases from 800 req/s to 1000 req/s, Figure 6.16c shows a substantial change in model placement. For some models like `resnet50`, adding a second replica becomes more attractive. At 800 req/s, the policy used one replica with batch size 4 and an approximate processing capability of 590 req/s. Any additional replica would only contribute 210 req/s, as the total processing capacity cannot exceed the target submission rate. However, at 1000 req/s, an additional replica may increase the total cluster goodput by 410 req/s. Consequently, the MILP solver begins to prefer smaller batch sizes with a higher replication factor. A similar analysis can be conducted for `densenet121`. With three replicas of batch size 8, `densenet121` has a comfortable buffer that allows it to accept some interference while most requests still meet their SLO.

Key Takeaway The WSM-MILP example highlights that any policy failing to model interference and queuing effects will prove ineffective as the system load increases. Once more, simply optimizing for total goodput and considering whether a model's expected goodput in isolation meets its target submission load is too simplistic and fails to take crucial factors into account. Higher inference latencies due to interference must be accounted for to assess whether a worker can still process its entire load within the SLOs. The examples of workers operating near maximum processing rate versus those with more buffer demonstrate how a worker can accommodate varying levels of interference while still providing SLO guarantees.

Observation 2: AO-MILP: Batch Size Selection Independent of Target Submission Rate

Observation Except for an SLO of 150ms, AO-MILP's total goodput shows a slight increase as the total system load rises. Comparing the model placements for AO-MILP in Figure 6.18 for tthe 3000 RPS and 5000 RPS scenario reveals interesting insights into batch size selection. We have chosen an SLO of 250ms to illustrate this behavior.



(a) RPS 3000 (b) RPS 5000

Figure 6.18: Model Placement for AO-MILP for an SLO of 250ms

AO-MILP selects a batch size of 64 for `densenet121` in both scenarios, independent of the target submission rate. When the target submission rate is 600 req/s per model, an average of 9.375 batches of size 64 can be built per second, each requiring an approximate build time of 107ms in the router, assuming no overheads and a constant arrival rate. This build time exceeds the `MAX_WAIT_TIME` constant set to 100ms in our system, suggesting that most batches would likely never reach their size of 64 and instead be sent to the node based on the timeout.

However, with a load of 1000 req/s, an average of 15.625 batches of size 64 can be built per second, again assuming no overheads and constant arrival rate. The batch build time in this case is approximately 64ms. Thus, we would expect most batches in this scenario to reach their desired size of 64. Figure 6.19 shows the batch sizes of `densenet121` over time for the 3000 and 5000 RPS scenario at an SLO of 250ms. The plot indeed validates our calculations.

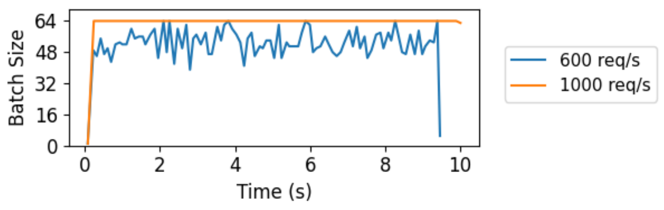


Figure 6.19: `densenet121` batch sizes over time at a target submission load of 600 req/s and 1000 req/s for AO-MILP

Key Takeaway The batch size selection should consider the model's request submission rate in conjunction with the `MAX_WAIT_TIME` threshold, as it can significantly impact how long requests spend at the router. Choosing an excessively large batch size may never be achieved if the request arrival rate is too low. Notably, neither the Usher implementations nor any of the MILP-based implementations account for this crucial aspect. They all assume that arrival rates for all models are always sufficient for all chosen batch sizes.

GPUs required by iGniter

We have previously already discussed iGniter aims to minimize the total number of GPUs required to fulfill the entire load. When the SLO is set to 350ms, iGniter requires 2, 3, 8, 8, and 8 GPUs for total loads of 1000, 2000, 3000, 4000, and 5000 requests per second, respectively. Consequently, we are only evaluating the placement of the first 4 GPUs in this case, providing a false impression on the effectiveness of iGniter's placement. However iGniter still fails to approach the ideal goodput for target submission loads of 1000 and 2000 requests per second where the GPU cluster size is sufficient. We refer the reader back to section 6.2.1 for the analysis, as the reasons are very similar.

Extending the Experiment to a Mix of Vision and Language Models

We have conducted the same experiment with a mix of vision and language models. Due to space constraints we do not discuss the results here but have added the resulting plots to the Appendix Section A.2.3.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This work introduced a novel system for scheduling machine learning (ML) inference workloads across a cluster of GPUs. The system's modular design allows for easy exchange of components, particularly the placement policy and workers, without necessitating changes to the entire architecture. This flexibility enabled us to evaluate various placement policies through a series of experiments, leading to the identification of several crucial aspects that cluster-level schedulers for ML inference must consider.

Our extensive examination of state-of-the-art works, such as Usher and iGniter, has allowed us to propose adjustments to address some of their shortcomings and identify new issues. The findings from this research lay the foundation for several critical decision points, each warranting deeper investigation. The key aspects are summarized below:

The correct Choice of Metrics Selecting the appropriate combination of metrics is crucial for quantifying a model's resource requirements. While we have focused extensively on compute metrics such as achieved occupancy and SM utilization, our work has revealed that a model's requirements cannot be reduced to a single compute and memory metric. While these are important, they are insufficient on their own. The challenge lies in identifying the right set of metrics to characterize a model comprehensively. Our review of related work indicates a lack of consensus in the research community regarding which metrics are most relevant, with different studies employing varied metrics tailored to their specific use cases.

Modelling Interference Many policies we reviewed model interference by summing up metrics that quantify a model's individual dependence on certain GPU components, ensuring the total does not exceed the GPU's capabilities. However, this approach is too simplistic. Profiling models in isolation and then simply merging their results is insufficient. The complex architecture of GPUs means that models may interfere at various levels, including also factors such as arrival rate distribution and total load to be served. More sophisticated models that combine and relate several of these layers are needed to accurately predict interference.

Modelling intra-cluster queuing Our research has demonstrated numerous instances where overall goodput was poor due to queuing at different parts of the cluster, particularly at the worker and router levels. These examples highlight the critical importance of avoiding queuing within the cluster, as it significantly degrades overall performance. Queuing can arise from various factors, including insufficient processing capacity, interference through colocation, or suboptimal cluster configuration. Any effective cluster scheduler must account for these situations, especially as system load increases.

The adequate choice of batch sizes Many issues we encountered were due to suboptimal batch size selection. While larger batch sizes can increase utilization and throughput, they also lead to longer inference latencies. In the context of inference, where requests must be served within strict Service Level Objectives (SLOs), batch sizes must be chosen carefully and cannot be arbitrarily large. This requirement significantly differentiates ML inference schedulers from those designed for training jobs, which are less latency-critical and primarily aim to maximize throughput.

Different notions of optimality While our work primarily focused on goodput as a metric for assessing placement policy effectiveness, other properties must also be considered. Optimizing solely for goodput in fixed-size clusters can lead to unfairness, favoring lighter models over heavier ones. However other criteria like fairness matter too as each user expects its model to be served.

All in all, our evaluations have demonstrated numerous examples where policies successfully colocated models for given loads and SLOs. This supports the conclusion that successful colocation is achievable, allowing for improved GPU utilization and cost reduction.

7.2 Future Work

Although we have not been able to present a novel placement policy, our system for scheduling ML inference workloads in GPU clusters and our findings lay the foundation for future research in this area. We identify several key directions for future work:

1. **Comprehensive Model Characterization**: A deeper understanding of model characteristics and corresponding quantitative metrics is crucial. An extensive analysis of the huge space of metrics trackable on NVIDIA GPUs would greatly benefit the research community, providing clarity on which metrics are most relevant beyond the space of creating effective placement policies.

2. **Modelling Interference**: Future placement policies should incorporate more sophisticated models of interference through colocation and the resulting queuing effects. This will require a multifaceted approach considering various GPU architectural aspects and workload characteristics.

3. **Dynamic Placement Policies**: Future work should consider dynamically integrating feedback from the workers. This could prove of great value in order to circumvent suboptimal placements. In addition it should be possible for clients to join and leave the cluster at different points in time.

4. **Extending the current system**: The current system is subject to several improvements. The networking architecture requires revision to enable direct workload transmission from clients to workers, rather than generating them at the worker-level. The clients should in addition become a separate entity living outside the cluster in order to model more realistic setups.

Appendix A

Appendix

A.1 Usher Profiling Results on a V100

Table A.1 shows the profiling profiling results for Usher on a V100 GPU attached to a `n1-standard-16` machine with the `Intel Skylake` CPU platform in Google Cloud Platform. Missing batch sizes are the result of out-of-memory errors. The table shows the columns

- **Mem Cap(%)**: is the maximum memory capacity reserved by a model expressed as a percentage from the total memory capacity of the GPU measured using `torch.cuda.max_memory_reserved` [42]
- **Ach Occ(%)**: is the achieved occupancy [21] measured using NVIDIA Nsight Compute (NCU)
- **wavg Ach Occ(%)**: is the weighted average achieved occupancy
- **wavg SM Util(%)**: is the weighted average SM utilization

Table A.1: Usher Profiling Results on a V100

Model	Batch Size	Latency (s)	Throughput (req/s)	Mem Cap(%)	Ach Occ(%)	wavg Ach Occ(%)	wavg SM Util(%)
alexnet	4	0.0014	2801.75	1.66	69.17	18.68	47.07
	8	0.0023	3540.12	2.08	85.97	29.47	83.79
	16	0.0031	5196.69	2.08	85.93	23.98	63.49
	32	0.0053	5990.46	2.80	89.94	36.71	96.24
	64	0.0097	6627.59	3.44	91.95	40.85	91.56
	128	0.0182	7023.69	6.40	93.02	45.96	99.00
bert	4	0.0341	117.34	3.47	85.89	31.83	96.82
	8	0.0658	121.53	4.06	88.46	32.63	98.92
	16	0.1281	124.88	5.25	91.29	33.72	99.51
	32	0.2439	131.19	7.63	92.90	27.22	99.77
	64	0.4853	131.88	12.40	93.83	27.47	99.82
	128	0.9647	132.68	21.92	94.21	27.61	99.89

Continued on next page

Table A.1: Usher Profiling Results on a V100

Model	Batch Size	Latency (s)	Throughput (req/s)	Mem Cap(%)	Ach Occ(%)	wavg Ach Occ(%)	wavg SM Util(%)
bloom_560	4	0.1400	28.58	19.95	93.20	40.35	94.72
	8	0.2728	29.33	26.29	93.80	41.26	97.66
	16	0.5452	29.35	43.73	94.11	42.50	98.82
densenet121	4	0.0154	260.13	0.57	84.97	8.20	13.90
	8	0.0169	472.72	0.93	87.28	12.37	21.26
	16	0.0192	832.59	1.55	90.79	31.83	54.10
	32	0.0335	954.82	2.84	93.61	49.97	86.92
	64	0.0629	1017.07	5.27	94.63	55.59	92.30
	128	0.1203	1063.81	10.12	94.77	59.10	97.81
efficientnet_b7	4	0.0299	133.93	2.45	94.14	13.92	22.47
	8	0.0308	260.14	3.54	92.96	28.25	43.55
	16	0.0465	344.10	5.69	96.78	52.48	81.32
	32	0.0883	362.31	9.94	98.06	60.03	92.35
	64	0.1609	397.70	18.43	98.41	62.05	95.81
	128	0.3153	405.93	35.43	99.33	64.23	98.86
gpt2	4	0.0369	108.28	5.80	91.28	42.51	97.73
	8	0.0732	109.23	10.48	93.05	54.08	98.84
	16	0.1435	111.49	19.85	93.84	55.29	99.55
	32	0.2730	117.21	38.58	94.27	43.27	99.77
inception_v3	4	0.0126	317.47	1.16	85.76	6.91	13.82
	8	0.0137	583.79	1.67	87.01	12.83	29.11
	16	0.0159	1004.73	3.07	90.82	19.85	47.39
	32	0.0271	1178.99	4.93	92.80	31.46	76.37
	64	0.0448	1427.86	8.61	92.77	37.40	88.81
	128	0.0813	1574.14	15.94	93.75	40.40	93.94
mobilenet_v2	4	0.0057	698.08	0.56	95.79	8.10	12.13
	8	0.0061	1322.24	1.03	89.73	14.62	22.44
	16	0.0066	2409.23	2.11	91.19	29.92	44.89
	32	0.0112	2855.66	4.25	92.62	61.22	88.65
	64	0.0211	3032.44	8.50	93.41	66.87	95.12
	128	0.0411	3117.90	17.00	93.80	69.11	97.73
resnet50	4	0.0068	589.78	1.16	87.39	17.55	36.26
	8	0.0096	829.08	1.77	90.83	34.36	70.49
	16	0.0160	998.99	2.70	92.63	45.14	87.33
	32	0.0300	1067.13	4.52	93.58	46.57	93.90
	64	0.0573	1117.12	8.16	93.95	48.55	98.04
	128	0.1113	1149.98	15.45	100.00	49.85	99.16
resnext50_32x4d	4	0.0098	408.34	1.04	87.43	17.86	36.12
	8	0.0148	540.23	1.61	90.80	35.33	68.28
	16	0.0234	684.65	2.54	92.81	43.66	82.91
	32	0.0420	762.39	4.36	93.54	44.80	91.70
	64	0.0780	820.18	8.00	98.80	48.18	96.52
	128	0.1505	850.23	15.29	100.00	49.45	98.43
t5	4	0.0311	128.74	3.17	97.18	43.39	77.25
	8	0.0580	137.83	4.88	97.49	50.87	94.80
	16	0.1096	146.02	8.15	97.74	53.50	97.78
	32	0.2131	150.19	16.08	97.79	56.72	98.84
	64	0.4211	151.97	29.17	97.89	58.93	99.41
vgg19	4	0.0098	408.51	4.19	92.15	33.91	95.18
	8	0.0182	438.65	5.40	93.07	34.86	98.88
	16	0.0262	610.11	10.58	93.56	48.88	99.27
	32	0.0516	620.69	18.50	93.77	53.35	99.57
	64	0.1021	627.01	34.32	93.88	55.28	99.69
	128	0.1994	641.97	65.91	100.00	47.67	99.76
xlnet	4	0.1088	36.77	5.22	95.76	44.47	70.97

Continued on next page

Table A.1: Usher Profiling Results on a V100

Model	Batch Size	Latency (s)	Throughput (req/s)	Mem Cap(%)	Ach Occ(%)	wavg Ach Occ(%)	wavg SM Util(%)
	8	0.2713	29.48	7.37	97.62	52.08	69.93
	16	0.6138	26.07	12.57	97.60	53.50	68.54
	32	1.2668	25.26	22.09	94.45	49.51	67.87
	64	2.5868	24.74	41.12	94.54	48.02	68.02

A.2 Additional Evaluation Plots

A.2.1 Limiting Number of GPUs with Vision Models without setting MPS shares at a target submission rate of 500 req/s

We have conducted the same experiment as described in Section 6.2.1, but do not set any MPS shares for Usher and WAO-Usher.

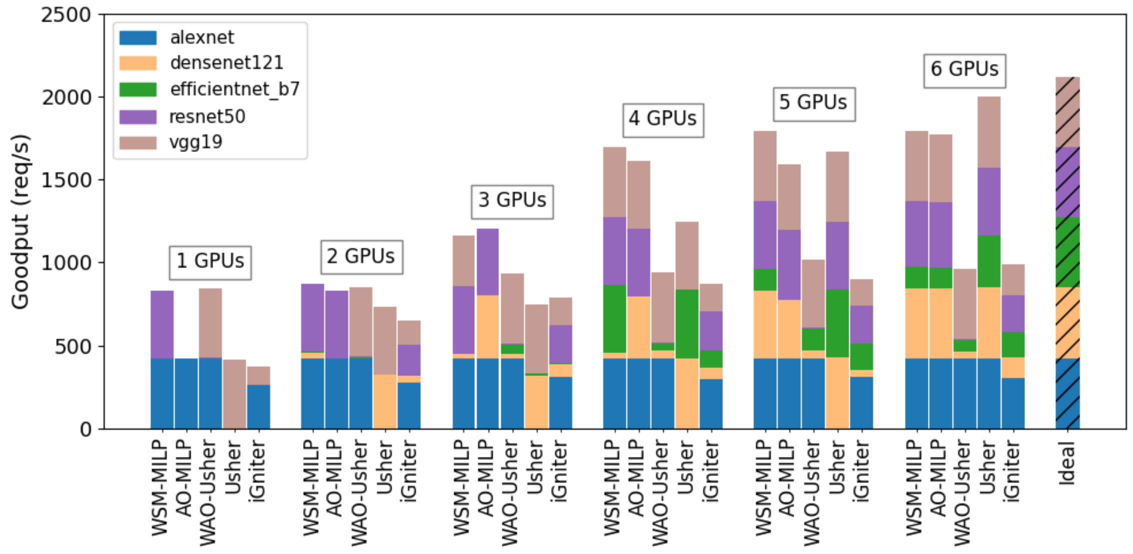


Figure A.1: Achieved Goodput in req/s for 5 vision models `alexnet`, `densenet121`, `efficientnet_b7`, `resnet50`, `vgg19` on a cluster of up to 6 NVIDIA V100 GPUs. Each model receives 4000 requests at target submission rate of 500 req/s. The SLO for each model is 200ms. No MPS share is set at the Workers for Usher and WAO-Usher.

A.2.2 Limiting the Number of GPUs with Vision Models and a Target Submission Rate of 1000 req/s

We have conducted the same experiment as described in Section 6.2.1, but instead have increased the target submission rate for each model to 1000 requests per second. The SLO is maintained at 200ms.

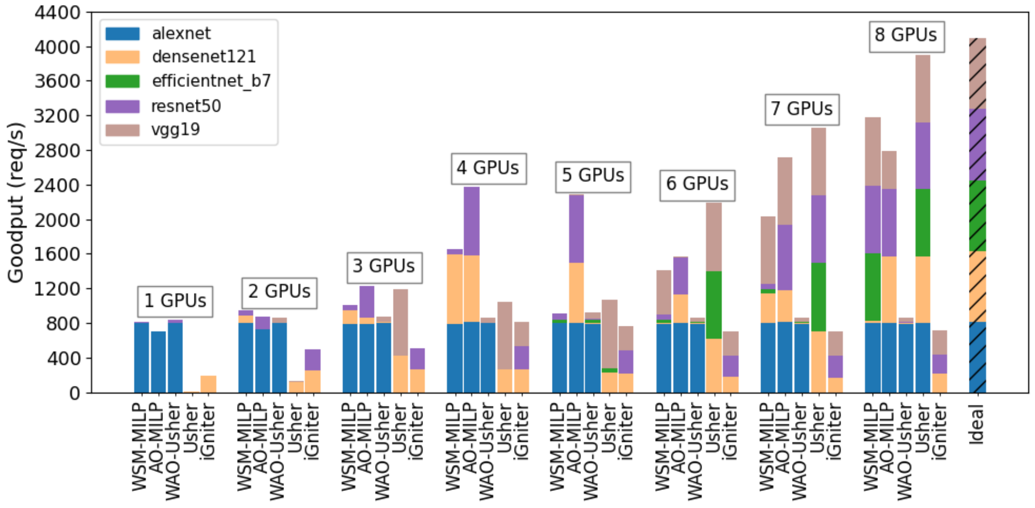
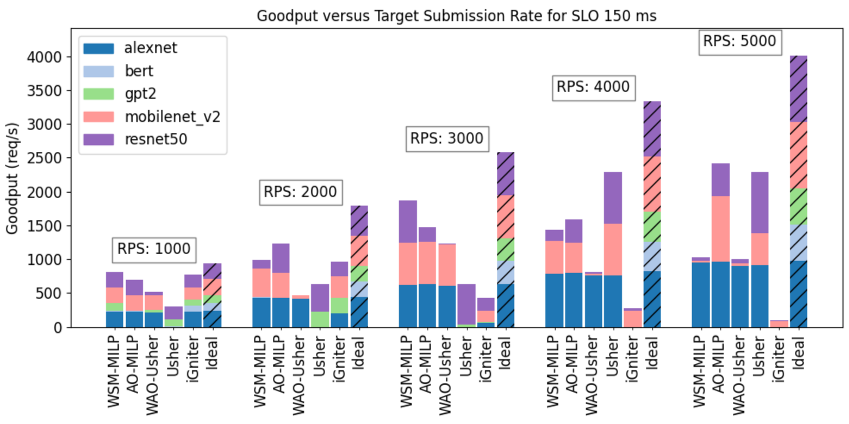


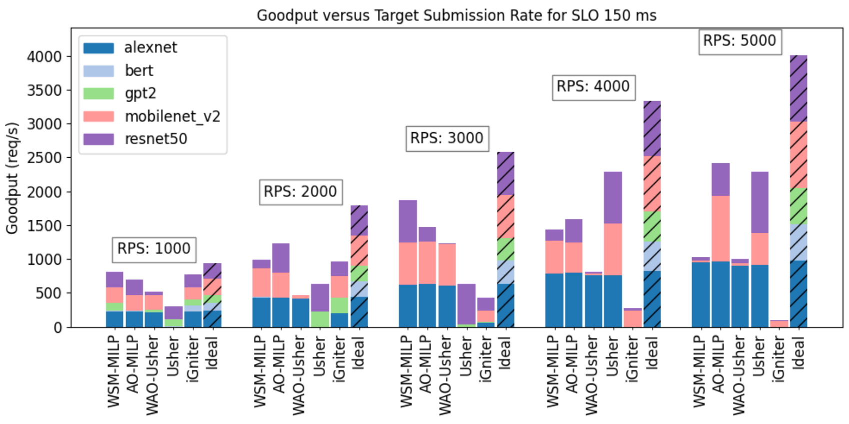
Figure A.2: Achieved Goodput in req/s for 5 vision models `alexnet`, `densenet121`, `efficientnet_b7`, `resnet50`, `vgg19` on a cluster of up to 8 NVIDIA V100 GPUs. Each model receives 8000 requests at a target submission rate of 1000 req/s. The SLO for each model is 200ms. MPS shares are set for Workers of Usher, WAO-Usher and iGniter.

A.2.3 Analysis of SLO and Target Load Variations for a Heterogeneous Model Mix

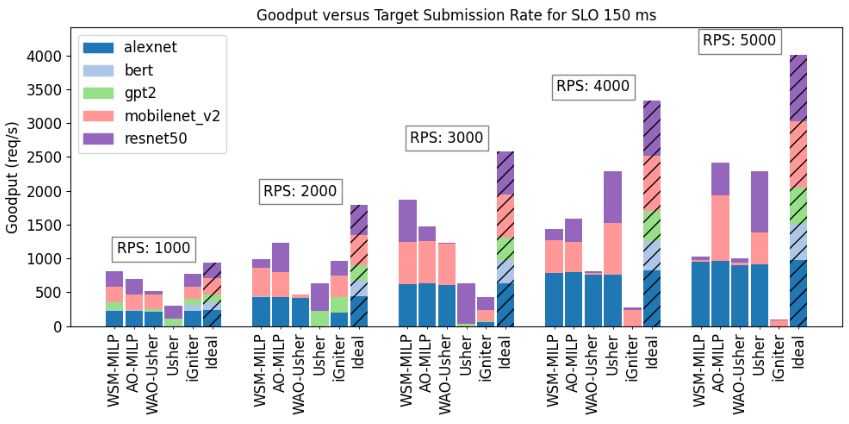
This section presents additional results for the experiment detailed in Section 6.3, focusing on a diverse mix of vision and language models. The model set includes `alexnet`, `bert`, `gpt2`, `mobilenet_v2`, and `resnet50`. We have allocated the total target load in a 2:1 ratio between vision and language models. Specifically, vision models receive twice the load of language models. The remaining experimental parameters and setup remain consistent with those described in the main text.



(a) SLO 150ms



(b) SLO 250ms



(c) SLO 350ms

Figure A.3: Goodput versus Target Submission Rate in a cluster of four NVIDIA V100 GPUs for a mix of vision and language models.

Bibliography

[1] Anthropic - claude 3 family. https://www.anthropic.com/news/claude-3-family.

[2] Bert for sequence classification. https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertForSequenceClassification.

[3] Bloom 560 for sequence classification. https://huggingface.co/docs/transformers/model_doc/bloom#transformers.BloomForSequenceClassification.

[4] Cuda compatibility matrix. https://docs.nvidia.com/deeplearning/cudnn/latest/reference/support-matrix.html#support-matrix.

[5] Cuda forward compatibility package. https://docs.nvidia.com/deploy/cuda-compatibility/#installing-the-forward-compatibility-package.

[6] Cuda programming model. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#programming-model.

[7] Deepspeed: Extreme speed and scale for dl training and inference. https://github.com/microsoft/DeepSpeed.

[8] Exponential distribution. https://en.wikipedia.org/wiki/Exponential_distribution.

[9] Github copilot. https://github.com/features/copilot.

[10] Google cloud platform n1 machine series. https://cloud.google.com/compute/docs/general-purpose-machines?hl=de#n1_machines.

[11] Google cloud platform: Virtual private cloud. https://cloud.google.com/vpc?hl=de.

[12] Google deepmind - gemini. https://deepmind.google/technologies/gemini/.

[13] Google protocol buffers. https://protobuf.dev/overview/.

[14] Google tensor processing units. https://cloud.google.com/tpu?hl=de.

[15] Gpt2 for sequence classification. https://huggingface.co/docs/transformers/model_doc/gpt2#transformers.GPT2ForSequenceClassification.

[16] Huggingface transformers. https://huggingface.co/docs/transformers/de/index.

[17] Igniter, an interference-aware gpu resource provisioning framework for achieving predictable performance of dnn inference in the cloud. github repo. https://github.com/icloud-ecnu/igniter.

[18] iperf3: A tcp, udp, and sctp network bandwidth measurement tool. https://github.com/esnet/iperf.

[19] Mapping metric from nvidia visual profiler to nvidia nsight compute. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html#metric-comparison.

[20] Networkx, minimum weighted maximum cardinality matching. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.matching.min_weight_matching.html.

[21] Nvidia achieved occupancy. https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm.

[22] Nvidia cuda streams. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[23] Nvidia multi-instance gpu. https://docs.nvidia.com/datacenter/tesla/mig-user-guide/.

[24] Nvidia multi-process service. https://docs.nvidia.com/deploy/mps/.

[25] Nvidia nsight compute not supported with mps. https://docs.nvidia.com/nsight-compute/ReleaseNotes/index.html#known-issues.

[26] Nvidia nsight compute profiler. https://developer.nvidia.com/nsight-systems.

[27] Nvidia nsight graphics profiler. https://developer.nvidia.com/nsight-graphics.

[28] Nvidia nsight systems profiler. https://developer.nvidia.com/nsight-systems.

[29] Nvidia system managment interface command line profiler. https://developer.nvidia.com/system-management-interface.

[30] Nvidia technical specifications. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications.

[31] Nvidia triton inference server. https://developer.nvidia.com/triton-inference-server.

[32] Nvidia visual profiler. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[33] Onnx, an open format built to represent machine learning models. https://onnx.ai.

[34] Openai, gpt-4. https://openai.com/index/gpt-4/.

[35] Openai, gpt-4o. https://openai.com/index/gpt-4o-and-more-tools-to-chatgpt-free/.

[36] Pulp, a linear and mixed integer programming modeler written in python. https://coin-or.github.io/pulp/.

[37] Python coroutines. https://docs.python.org/3/library/asyncio-task.html.

[38] Python grpc. https://grpc.github.io/grpc/python/.

[39] Python grpc asyncio api. https://grpc.github.io/grpc/python/grpc_asyncio.html.

[40] Python managers. https://docs.python.org/3/library/multiprocessing.html#managers.

[41] Pytorch alexnet. https://pytorch.org/vision/stable/models/alexnet.html.

[42] Pytorch cuda max memory reserved. https://pytorch.org/docs/stable/generated/torch.cuda.max_memory_reserved.html#torch.cuda.max_memory_reserved.

[43] Pytorch densenet121. https://pytorch.org/vision/stable/models/generated/torchvision.models.densenet121.html#torchvision.models.densenet121.

[44] Pytorch efficientnet b7. https://pytorch.org/vision/stable/models/generated/torchvision.models.efficientnet_b7.html#torchvision.models.efficientnet_b7.

[45] Pytorch inception v3. https://pytorch.org/vision/stable/models/generated/torchvision.models.inception_v3.html#torchvision.models.inception_v3.

[46] Pytorch mobilenet v2. https://pytorch.org/vision/stable/models/generated/torchvision.models.mobilenet_v2.html#torchvision.models.mobilenet_v2.

[47] Pytorch pined memory. https://pytorch.org/tutorials/intermediate/pinmem_nonblock.html.

[48] Pytorch resnet50. https://pytorch.org/vision/stable/models/generated/torchvision.models.resnet50.html#torchvision.models.resnet50.

[49] Pytorch vgg19. https://pytorch.org/vision/stable/models/generated/torchvision.models.vgg19.html#torchvision.models.vgg19.

[50] Pytorch vision models. https://pytorch.org/vision/stable/models.html.

[51] Rayserve: Scalable and programmable serving. https://docs.ray.io/en/latest/serve/index.html.

[52] T5 for sequence classification. https://huggingface.co/docs/transformers/model_doc/t5#transformers.T5ForSequenceClassification.

[53] Tensorrt, an ecosystem of apis for high-performance deep learning inference. https://developer.nvidia.com/tensorrt.

[54] Tensort command-line wrapper. https://github.com/NVIDIA/TensorRT/tree/main/samples/trtexec.

[55] Triton model configuration - dynamic batcher. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/user_guide/model_configuration.html#dynamic-batcher.

[56] Usher github repository. https://github.com/ss7krd/Usher.

[57] Xlnet for sequence classification. `https://huggingface.co/docs/transformers/model_doc/xlnet#transformers.XLNetForSequenceClassification`.

[58] Saurabh Agarwal, Amar Phanishayee, and Shivaram Venkataraman. Blox: A modular toolkit for deep learning schedulers. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 1093–1109, 2024.

[59] Yehia Arafa, Abdel-Hameed Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. Low overhead instruction latency characterization for nvidia gpgpus, 05 2019.

[60] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on {Multi-GPU} servers with {Spatio-Temporal} sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 199–216, 2022.

[61] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A {Low-Latency} online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[62] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. Ebird: Elastic batch for improving responsiveness and throughput of deep learning services. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*, pages 497–505. IEEE, 2019.

[63] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, et al. Enable simultaneous dnn services based on deterministic operator overlap and precise latency prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

[64] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020.

[65] Strati Foteini, Ma Xianzhe, and Klimovic Ana. Orion: Interference-aware, fine-grained gpu sharing for ml applications. Association for Computing Machinery, 2024.

[66] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving {DNNs} like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 443–462, 2020.

[67] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, 2018.

[68] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. Dynamic space-time scheduling for gpu inference. *arXiv preprint arXiv:1901.00041*, 2018.

[69] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[70] Baolin Li, Siddharth Samsi, Vijay Gadepally, and Devesh Tiwari. Clover: Toward sustainable ai with carbon-aware machine learning inference service. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2023.

[71] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E Gonzalez, et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 663–679, 2023.

[72] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. Interference-aware scheduling for inference serving. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, pages 80–88, 2021.

[73] Nathan Otterness, Ming Yang, Sarah Rust, Eunbyung Park, James H. Anderson, F. Donelson Smith, Alex Berg, and Shige Wang. An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 353–364, 2017.

[74] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. An analysis of collocation on gpus for deep learning training. In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pages 81–90, 2024.

[75] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. {INFaaS}: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 397–411, 2021.

[76] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. USHER: Holistic interference avoidance for resource optimized ML inference. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 947–964, Santa Clara, CA, July 2024. USENIX Association.

[77] C Tan, Z Li, J Zhang, Y Cao, S Qi, Z Liu, Y Zhu, and C Guo. Serving dnn models with multi-instance gpus: A case of the reconfigurable machine scheduling problem (2021). *arXiv preprint arxiv:2109.11067*.

[78] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[79] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. Morphling: Fast, near-optimal auto-configuration for cloud-native model serving. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 639–653, 2021.

[80] Xiaorui Wu, Hong Xu, and Yi Wang. Irina: Accelerating dnn inference with efficient online scheduling. In *Proceedings of the 4th Asia-Pacific Workshop on Networking*, pages 36–43, 2020.

[81] Fei Xu, Jianian Xu, Jiabin Chen, Li Chen, Ruitao Shang, Zhi Zhou, and Fangming Liu. igniter: Interference-aware gpu resource provisioning for predictable dnn inference in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):812–827, 2022.

[82] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. {SHEPHERD}: Serving {DNNs} in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 787–808, 2023.