Helix: Distributed Serving of Large Language Models via Max-Flow on Heterogeneous GPUs

Yixuan Mei
Carnegie Mellon University

Yonghao Zhuang
Carnegie Mellon University

Xupeng Miao
Carnegie Mellon University

Juncheng Yang
Carnegie Mellon University

Zhihao Jia
Carnegie Mellon University

Rashmi Vinayak
Carnegie Mellon University

Abstract

This paper introduces Helix, a distributed system for high-throughput, low-latency large language model (LLM) serving on heterogeneous GPU clusters. A key idea behind Helix is to formulate inference computation of LLMs over heterogeneous GPUs and network connections as a *max-flow* problem for a directed, weighted graph, whose nodes represent GPU instances and edges capture both GPU and network heterogeneity through their capacities. Helix then uses a mixed integer linear programming (MILP) algorithm to discover highly optimized strategies to serve LLMs. This approach allows Helix to jointly optimize model placement and request scheduling, two highly entangled tasks in heterogeneous LLM serving. Our evaluation on several heterogeneous cluster settings ranging from 24 to 42 GPU nodes show that Helix improves serving throughput by up to 2.7× and reduces prompting and decoding latency by up to 2.8× and 1.3×, respectively, compared to best existing approaches.

1 Introduction

Generative large language models (LLMs) such as GPT-4 [1] and LLaMA-3 [20] have demonstrated exceptional capabilities of creating natural language texts across a spectrum of application domains, including chatbot [25], coding assistant [19, 30], and task automation [11]. However, the increasingly large model sizes and high computational requirements of modern LLMs make it challenging to serve them cheaply and efficiently on modern cloud platforms. In particular, most of today's LLM serving systems (e.g., Orca [46] and vLLM [16]) target *homogeneous* GPU clusters [21], where all GPUs are of the same type and have identical memory capacity and compute resources. Due to the increasing model sizes, serving LLMs using homogeneous GPUs requires an increasing number of GPUs, as shown in Table 1. In addition, serving state-of-the-art LLMs used in industry requires even more resources. We observed it's nearly infeasible to allocate GPUs of this magnitude within a single cloud region, which is also acknowledged by other recent works [36, 45].

Table 1: Minimum numbers of GPUs required to serve LLMs in existing homogeneous serving systems. We use half of GPU memory to store model parameters and the other half for key-value cache.

LLMs	Num. of Parameters	Num. of L4s	Num. of A100s	Num. of H100s
LLaMA-3 [41]	70 billion	12	7	4
GPT-3 [1]	175 billion	30	18	9
Grok-1 [44]	314 billion	53	32	16

Due to advances in GPU architectural designs and the incremental deployment of them over time, modern cloud platforms increasingly consist of a mix of GPU types. These *heterogeneous* GPU instances are spread to datacenters around the world and collectively offer significantly larger memory capacity and more compute resources than individual GPU types, enabling a more accessible and scalable approach to LLM serving. Similarly, there is also a trend of using volunteer consumer GPUs to address the GPU scarcity problem [7, 32, 47]. However, in contrast to homogeneous GPU instances, deploying LLMs on geo-distributed heterogeneous instances necessitates accommodating various GPU devices and network conditions.

Prior work has introduced several systems for running machine learning (ML) computation over heterogeneous devices [26, 49] or geo-distributed environments [12, 31]. But most of them are designed for long-running training workloads and cannot adapt to LLM serving scenarios with real-time inference requests. For example, the most closely relevant work to ours is Petals [4], which uses *pipeline model parallelism* to partition an LLM into multiple stages and employs a greedy algorithm to assign heterogeneous GPUs to these stages in a round-robin fashion. Such throughput-optimized design is effective for long-duration tasks but lacks the responsiveness required for interactive LLM applications like chatbot that demand rapid responses.

Correspond to: Yixuan Mei <yixuanm@andrew.cmu.edu>

To leverage heterogeneous GPU resources for LLM serving, we propose Helix, a distributed system that enables high-throughput, low-latency LLM serving on heterogeneous GPUs. The key idea behind Helix is to formulate the execution of LLM serving over heterogeneous GPUs and networks as the *data flow* problem under the constraints of diverse GPU computing capabilities, memory capacities, as well as complex inter-GPU connections. To the best of our knowledge, Helix is the first serving system designed for geo-distributed heterogeneous GPUs. Next, we introduce the key challenges Helix must address and Helix's solutions to them.

First, due to the increasing size of LLMs, serving them on modern GPUs requires employing *tensor* [35] and *pipeline* [13, 26] model parallelism to partition an LLM into multiple stages and place these stages on different GPUs, a task we term *model placement*. Homogeneous serving systems (e.g., Orca [46]) partition an LLM into equal-sized stages and assign them to GPUs. This approach results in suboptimal utilization of high-performance GPUs as it accommodates the memory and computational limitations of less powerful GPUs. Existing heterogeneity-aware serving systems (e.g., Petals [4]) rely on different heuristics to partition a model into stages and assign them to GPUs. Existing heuristics do not simultaneously consider both GPU and network heterogeneity.

将 model placement 表述为一个最大流问题，并用一个 MILP 算法来求解

Helix formulates model placement as a *max-flow* problem of a directed, weighted graph, whose nodes represent GPU instances and edges capture both GPU and network heterogeneity through their capacities in the max-flow problem. Helix then uses a mixed integer linear programming (MILP) algorithm to discover highly optimized model placement strategies, which largely outperform the heuristic methods used in prior work [4].

A second challenge Helix must address is *request scheduling*. To serve an LLM request, Helix needs to select a *pipeline* of GPU instances to compute all layers of the LLM. Existing systems generally employ a group of fixed pipelines and assign requests to these pipelines in a round-robin fashion. Using fixed pipelines is not flexible enough to accommodate the heterogeneous compute and network conditions and often causes under-utilization. Instead, Helix introduces *per-request pipelines*, where each request is assigned its own pipeline. As a result, the total number of potential pipelines is equal to the number of paths from source to sink in the graph representation of the cluster, which offers sufficient flexibility for Helix to maximally utilize the full capacity of GPU instances and network connections between them.

已有工作一般用固定的 GPU instance pipeline 来服务 requests，这里选择 per-request pipeline，存储 stage i 的 instance 可以 route 到任意一个存储 stage i+1 的 instance，即 pipeline 数目等于集群图表示中所有从源到终点的路线数目

We have implemented Helix on top of vLLM [16] and evaluated it on three heterogeneous clusters ranging from 24 to 42 nodes, with up to 7 different node types. The models we evaluated include LLaMA 30B and 70B. Compared to heterogeneity-aware baselines, Helix improves serving throughput by up to 2.7× while reducing average prompting and decoding latency by up to 2.8× and 1.3×, respectively.

In summary, this paper makes the following contributions: (1) the first system for high-throughput, low-latency LLM serving on heterogeneous GPUs, (2) a max-flow formulation for LLM serving and an MILP algorithm to optimize model placement; (3) per-request pipelines to maximize GPU utilization; and (4) an implementation of our techniques and an evaluation on various LLM benchmarks. We will release Helix after paper review to facilitate future research.

2 Background

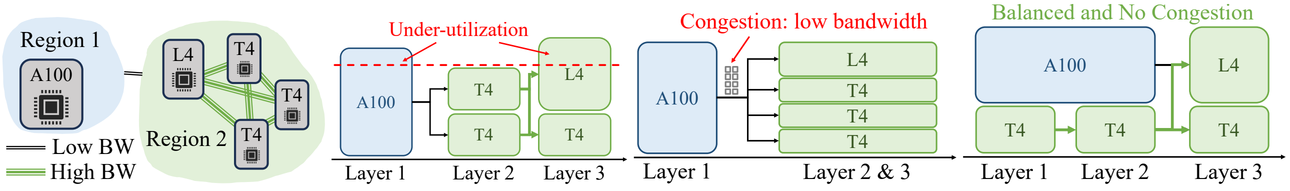
2.1 LLM Architecture and Serving

Most of today's LLMs adopt a decoder-only Transformer architecture [5, 29], which begins by converting a natural language query into a sequence of tokens. The model then converts each token into a hidden state vector, whose size is referred to as the model's hidden size. A Transformer model comprises of input and output embeddings and a series of identical Transformer layers, each consisting of a self-attention and a feed-forward block. A self-attention block calculates the 'affinity' between every pair of tokens and updates each token's hidden states based on this contextual relevance score. Feed-forward blocks independently modify each token's hidden state through a non-linear function.

Given an input sequence, a Transformer model computes the probability distribution for the next token, and samples from this probability distribution. Thus, the model applies an auto-regressive paradigm to generate the whole output sequence: given an *input prompt*, a model runs multiple iterations. At the first iteration, as known as the *prompt phase*, the model processes all prompt tokens and generates the first output token. In subsequent iterations, as known as the *decode phase*, the model incorporates both prompt and previously generated tokens to predict the next output token. This iterative process stops when model produces a special end-of-sentence signal ($\langle$eos$\rangle$). Since the generation output is unpredictable, the exact number of iterations remains uncertain until the sequence is fully generated.

In addition to the unpredictable execution iterations, another feature of LLM serving is the high memory demand. The self-attention block requires all previous tokens' hidden states as inputs. To store the hidden states (known as the KV-cache) for newly generated tokens, the memory requirements keep increasing along the generation process.

To address these challenges, Orca [46] presented iteration-level scheduling, which updates a batch at every iteration to avoid resource retention when a request is completed but others in the same batch need more iterations; vLLM [16] introduces PagedAttention, managing memory for KV-cache with identical pagesand allocating a new page only when a request has used up all its pages; multi-query [33] and group-query attention [3] modifies the self-attention mechanism to reduce the size of KV-cache stored for each token.



(a) Five nodes in two clusters. (b) Uniform model partition. (c) Balanced FLOPs. (d) Network aware placement.

Figure 1: Examples of sub-optimal model placement and request schedule. 1a) all GPUs and network condition in this example. The order of compute capacity is: A100 > L4 > T4; 1c) Model placement by uniformly partition the model, then allocate devices by a balanced compute capacity; 1b) Co-optimizing model partition and device placement to make the compute capacity more balanced; 1d) Co-optimizing model partition, device placement, and request scheduling.

2.2 Distributed Model Serving

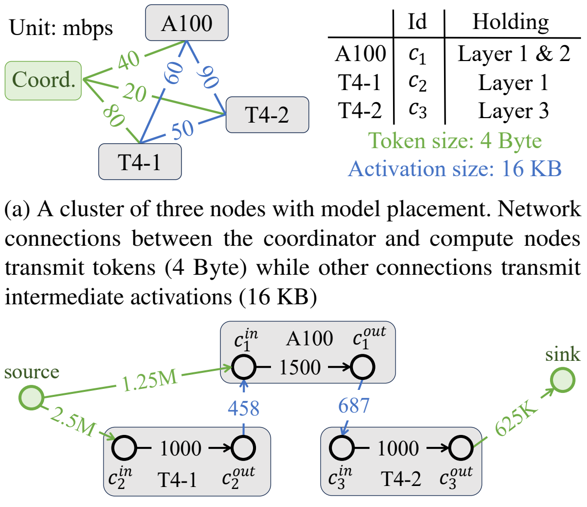
Open source LLMs now feature up to hundreds of billions of parameters, far exceeding the memory capacity of a single GPU. Consequently, serving an LLM requires multiple GPUs operating in parallel. Tensor Parallelism (TP) [35] partitions the weight of each operator among GPUs, gathering the partial results on each device via an AllReduce/AllGather operation. However, TP is highly sensitive to network conditions. For every Transformer layer, it needs two communications. As a result, TP has a significant overhead in high-latency networks, and is only used among GPUs within a node.

Conversely, Pipeline Parallelism (PP) [13] assigns different operators (typically multiple layers) across GPUs to create multiple pipeline stages. It then splits inputs into micro-batches, running them through the pipeline. PP only transmits the activation tensor at the boundary of pipeline stages. Hence, PP is much less network-sensitive. However, it is challenging to perfectly partition both the model and input batch, which results in pipeline bubbles. As a result, PP suffers from the device idle at pipeline bubbles, necessitating careful schedule to be performant [2].

Traditional data center setups typically assume *homogeneous* clusters: uniform nodes with a uniform bandwidth. As a result, models are *evenly* partitioned into pipeline stages and assigned to each device. However, with the growing size of LLM to be served and the shortage of the latest generation GPUs, serving LLM with heterogeneous devices emerges as a critical demand, which is not well studied in previous efforts.

Serving LLMs on heterogeneous GPUs needs to consider heterogeneity for both hardware and network. For example, evenly partition of model layers may under-utilizates more capable devices. Additionally, nodes may be located in multiple zones. The bandwidth discrepancies between different zones must be considered.

To tackle these challenges, Petals [4] introduced a greedy device allocation and request routing. It focused on a decentralized setup, which allows hot plug-in and -out of devices. For device placement, it allocates new devices to the least efficient pipeline stages. For network topology, it routes each request by prioritizing the low-latency network. However, such a greedy online approach is sub-optimal in the heterogeneous cluster scenario, where the information of all nodes is a known priori. SWARM [31] focused on model training in a heterogeneous cluster. It evenly partitions the model into pipeline stages. When routing a request to the next pipeline stage, it selects the replica based on the real-time throughput of each candidate.



	Id	Holding
A100	c_1	Layer 1 & 2
T4-1	c_2	Layer 1
T4-2	c_3	Layer 3

(a) A cluster of three nodes with model placement. Network connections between the coordinator and compute nodes transmit tokens (4 Byte) while other connections transmit intermediate activations (16 KB)

(b) Graph abstraction of the cluster.

Figure 2: Graph abstraction of a 3-node cluster with given model placement. Numbers on the edges in Fig. 2b represent their capacity, which is the number of tokens that can pass through the edges per second. Max flow between source and sink equals the max serving throughput of the cluster.

3 Optimization Formulation in Helix

This section first provides a mathematical abstraction for LLM serving systems. Based on this formulation, we model heterogeneous LLM serving as a Max-Flow problem. Finally, we apply mixed-integer linear programming (MILP) to search for a model placement strategy with the highest max flow.

3.1 Formulation of LLM Serving

A cluster to serve LLMs generally contains one coordinator node h and a group of compute nodes $\mathcal{C}$. Each compute node $c_i \in \mathcal{C}$ has a compute throughput T_i and given GPU VRAM size. Network connections between nodes in the cluster have given their throughput and latency. Based on the cluster information, LLM serving requires finding a placement of model layers to compute nodes to maximize *serving performance*. The model placement function $\Psi : \mathcal{C} \mapsto \mathcal{P}(\mathcal{M})$ takes as input a compute node and returns a (usually continuous) subset of the model $\mathcal{M}$. One widely used metric [4, 31] to assess a model placement is the performance of the pipeline stage with the lowest compute capacity $\min_i \sum_j \text{capacity}(j) \cdot 1_{i \in \Psi(j)}$, which sums up the compute capacity of all instances serving one layer. As we will show below, only considering compute capacity yields sub-optimal model placements for heterogeneous clusters.

Based on model placement, LLM serving requires a request scheduling strategy that can efficiently serve requests in the cluster. The request scheduling strategy $\phi : \mathcal{R} \mapsto \mathcal{C}^k$ inputs a request and outputs a sequence of compute nodes that form a complete pipeline for executing all layers of the LLM.

Before diving into our Max-Flow formulation of heterogeneous LLM serving, we first use an example to show why we need to co-optimize model partition, device placement, and request scheduling in a Max-Flow problem.

In this example, the devices are shown in Fig. 1a. There are two regions with a low bandwidth between them. Region 1 has a powerful A100 GPU, while Region 2 has a less powerful L4 GPU and three T4 GPUs, but has a high bandwidth within the region. The pairwise bandwidths are independent. If we, following the common approach, first partition the model, then assign devices to each partition, the placement plan will be as Fig. 1b. In this plan, although the last pipeline stage has a T4 and an L4 GPU, its throughput is bound by the previous stage's output throughput, which only has 2 T4 GPUs. This indicates a necessity to co-optimize the pipeline partition plan and placement of pipeline stages.

However, even with a perfectly balanced compute capacity at each pipeline stage, as shown in Fig. 1c, the solution can still be sub-optimal. In this solution, it assigns the powerful A100 to individually serve some layers, while other GPUs runs in a data parallel manner for the rest layers. However, communications from one pipeline stage to another become the bottleneck. For every request, its intermediate state is sent from Region 1 to Region 2 via low bandwidth. This eventually creates congestion on the A100's send side. Instead, Fig. 1d assigns two T4 GPUs running in parallel with the A100. This splits the workload between the A100 GPU and the two T4 GPUs, so as to reduce the communication volume on the slow link.

3.2 Heterogeneous LLM Serving as Max-Flow

To optimize model placement, Helix needs a way to determine the max serving throughput of different model placements. To achieve this, we transform a cluster of compute nodes with assigned model layers into a directed graph with edge capacity. The edge capacity denotes the number of tokens compute nodes and network connections can process and transmit per second. Max flow between source and sink vertices in the graph, which represent the coordinator node, gives us the max serving throughput of the cluster with current model placement. Below we show the formal construction of a cluster's graph abstraction with a given model placement.

For a given cluster with coordinator node h, a set of compute nodes $\mathcal{C}$, and a model placement Ψ, we can transform entities in the cluster into elements of its graph abstraction as follows. An example of such a graph abstraction of a cluster with *given model placement* is shown in Fig. 2

Compute Nodes. For each compute node $c_i \in \mathcal{C}$, we represent it with two connected vertices in the graph. We name the two vertices c_i^{in} and c_i^{out}. The capacity of the directed edge (c_i^{in}, c_i^{out}) represents the max number of tokens this node can process within one second. It is the minimum of the node's compute and network throughput. Helix performs a one-time profiling to measure the throughput of all compute nodes.

Coordinator Node. We represent the coordinator node as source and sink vertices in the graph.

Network Connection. In a given cluster, a node may communicate with any other nodes, creating $O(|\mathcal{C}|^2)$ possible directed network connections between different nodes. However, only a subset of those connections are valid based on the model placement as described below. A *valid connection* should satisfy one of the following three criteria: (1) the connection is from coordinator node h to compute node c_i and c_i holds the first layer of the model; (2) the connection is from a compute node c_j to coordinator node h and c_j holds the last layer of the model; (3) the connection is from one compute node c_i to another compute node c_j and c_j holds model layers immediately needed after inference on c_i. For the first and second case, we represent the connection with directed edge $(source, c_i^{in})$ and $(c_j^{out}, sink)$ respectively, with capacity equal to the connection bandwidth divided by the transmission size of a token (a few bytes). For the third case, we represent the connection with a directed edge (c_i^{out}, c_j^{in}), and the capacity equals to the connection bandwidth divided by the transmission size of an activation (tens of kilobytes). This models the throughput constraint imposed by network connection speed between different nodes. We denote the full set of possible network connections as $\mathcal{E}$.

After constructing the equivalent graph abstraction of a cluster, we run preflow-push algorithm [6] to get the max flow between source and sink node. One unit of flow here represents one token that can pass through a compute node or

Table 2: Variables used in MILP.

Symbol	Type	Num.	Description
s_i	int	$O(\|\mathcal{C}\|)$	index of c_i's first layer
b_i^j	binary	$O(\|\mathcal{C}\|)$	whether c_i holds j layers
$f_{i,j}$	real	$O(\|\mathcal{E}\|)$	flow from c_i to c_j
$d_{i,j}$	binary	$O(\|\mathcal{E}\|)$	whether (c_i, c_j) is valid
$cond_{i,j}^1$	binary	$O(\|\mathcal{E}\|)$	aux. variable in constraint-4
$cond_{i,j}^2$	binary	$O(\|\mathcal{E}\|)$	aux. variable in constraint-4

Table 3: Constraints used in MILP.

Group	Num.	Constraint
Model placement	$O(\|\mathcal{C}\|)$	$\sum_{j=1}^k b_i^j = 1$
	$O(\|\mathcal{C}\|)$	$0 \leq s_i < L$ and $e_i \leq L$
Flow conservation	$O(\|\mathcal{C}\|)$	$\sum_u f_{u,i} = \sum_v f_{i,v}$
Infer. throughput	$O(\|\mathcal{C}\|)$	$\sum_u f_{ui} \leq \sum_{j=1}^k b_i^j \cdot \text{throughput}_j$
Connection validity	$O(\|\mathcal{C}\|)$	$s_i \leq L \cdot (1 - d_{source,i})$
	$O(\|\mathcal{C}\|)$	$L \cdot d_{i,sink} \leq e_i$
	$O(\|\mathcal{E}\|)$	$(L+1)(1 - cond_{i,j}^1) \geq s_j - e_i$
	$O(\|\mathcal{E}\|)$	$e_j - e_i \geq 1 - (L+1)(1 - cond_{i,j}^2)$
	$O(\|\mathcal{E}\|)$	$d_{i,j} \leq 0.5 * cond_{i,j}^1 + 0.5 * cond_{i,j}^2$
Trans. throughput	$O(\|\mathcal{E}\|)$	$f_{i,j} \leq d_{i,j} \cdot \text{throughput}_{i,j}$

network connection in one second. Therefore, the max flow gives us the max possible serving throughput of the cluster with current model placement.

3.3 Optimal Model Placement with MILP

The previous section presented an approach for obtaining the max serving throughput of a cluster *for a given model placement*. In this section, we introduce a mixed-integer linear programming (MILP)-based method to find a model placement that maximizes the max flow, thus maximizing serving throughput. The MILP formulation has linear number of variables and constraints with respect to the number of compute nodes and network connections. Key challenges addressed include (1) formulation of system-level constraints as linear number of conditions to satisfy, (2) expression of these conditions with linear number of variables, and (3) linearization of each condition using auxiliary variables, specifically, each constraint is expressed as most three linear constraints with the help of at most two auxiliary variables. An overview of the variables and constraints is shown in Table. 2 and 3.

Node variables. To represent the model placement on each compute node, we introduce two groups of variables in our MILP formulation. Suppose the model has a total of L layers and each compute node holds a continuous subset of the model. For each compute node c_i, we introduce an integral variable s_i to represent the first layer c_i holds. Suppose compute node c_i can hold at most k layers on its GPU, we further introduce k binary variables $b_i^1, b_i^2, ..., b_i^k$ to indicate the number of layers node c_i holds ($b_i^j = 1$ if c_i holds j layers). We choose to express model placement with k binary variables (instead of one integer for the number of layers) because this formulation facilitates the expression of inference throughput constraints as discussed below. The *end layer index* of c_i can be expressed as $e_i = s_i + \sum_{j=1}^k j \cdot b_i^j$. Therefore, c_i holds layers in range $[s_i, e_i)$.

Connection variables. We introduce two groups of variables to constrain the number of inference requests that can go through each network connection in the cluster. For network connection between compute node c_i and c_j, we introduce a real variable $f_{i,j}$ to denote the amount of flow from c_i^{out} to c_j^{in} in the graph abstraction. We further introduce a binary variable $d_{i,j}$ to denote whether the network connection is valid (as defined in Sec. 3.2). The constraints we introduce below will use $d_{i,j}$ to ensure that requests can only be transmitted through valid connections. For network connections between coordinator node and compute nodes, we similarly introduce two variables as above, but replace i/j with source/sink.

Constraint-1: model placement. To ensure that the model placement found by the MILP solver is valid, we need the following two constraints for each compute node c_i. First, c_i should have only one valid model placement, meaning that $\sum_{j=1}^k b_i^j = 1$. Moreover, the first and last layer c_i holds must be within the range of L layers, meaning that $0 \leq s_i < L$ and $e_i \leq L$.

Constraint-2: flow conservation. For each compute node c_i, the sum of flow that goes in to c_i^{in} must equal to that goes out of c_i^{out} because of flow conservation. This constraint can be expressed as $\sum_u f_{u,i} = \sum_v f_{i,v}$, where u and v enumerates through all nodes except i.

Constraint-3: inference throughput. For compute node c_i, the amount of flow that passes through (c_i^{in}, c_i^{out}) should be no larger than its maximum inference throughput. We can impose this constraint with $\sum_u f_{ui} \leq \sum_{j=1}^k b_i^j \cdot T_j$. Here, T_j is a constant that represents the maximum number of tokens node c_i can process in one second when holding j layers, which is obtained through a one-time profiling process.

Constraint-4: connection validity. We need to determine the validity of network connections to know whether requests can be transmitted through them. For a network connection from the coordinator node to compute node c_i, it is valid only if c_i holds the first layer of the model. To express this constraint with MILP, we need to linearize it into the following form: $s_i \leq L \cdot (1 - d_{source,i})$. Similarly, for network connection from compute node c_i to coordinator, we constrain its validity with $L \cdot d_{i,sink} \leq e_i$. For network connection from compute node c_i to c_j, its validity $d_{i,j}$ is determined by whether $s_j \leq e_i < e_j$ holds. To linearize this condition, we need to introduce two binary auxiliary variables $cond_{i,j}^1$ and $cond_{i,j}^2$. $cond_{i,j}^1$ takes value 1 only if $s_j \leq e_i$, which can be linearized as $(L+1)(1 - cond_{i,j}^1) \geq s_j - e_i$. $cond_{i,j}^2$ takes value 1 only if $e_i < e_j$,

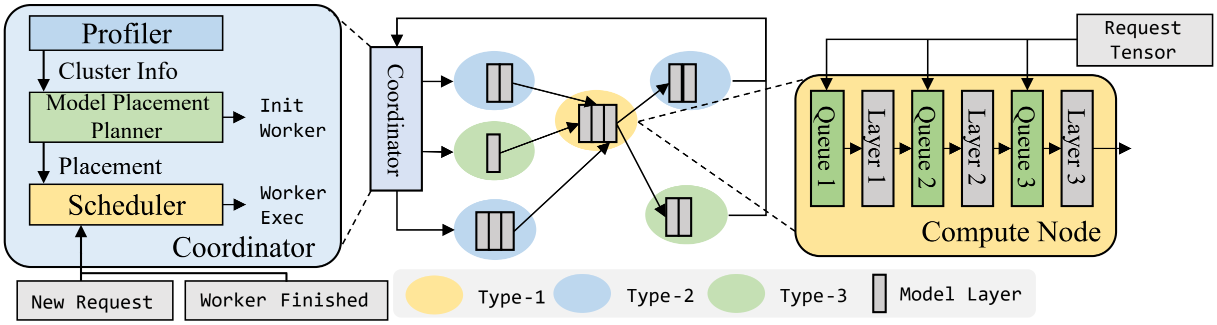


Figure 3: **Helix overview.** In Helix, the coordinator plans model placement as described in Sec. 3.3. We only need to run model placement once for each cluster. When a new request arrives, the coordinator node runs Helix scheduler to assign it a per-request pipeline and sends the request to the first node in the pipeline. Each compute node in the pipeline performs inference on the request on the layers it is responsible for and sends the (output for the) request to the next node in the pipeline. When the last node in the pipeline finishes performing inference on its layers, it will send the output token for the request to the coordinator (Worker Finished). The coordinator schedules generation of the next token for the request using the same pipeline.

which can be linearized as $e_j - e_i \geq 1 - (L+1)(1 - cond^2_{i,j})$. The network connection is valid only if both binary auxiliary variables holds true, which can be expressed as $d_{i,j} \leq 0.5 * cond^1_{i,j} + 0.5 * cond^2_{i,j}$.

We remark that if $s_j < e_i < e_j$, then requests coming from c_i will only infer layers $[e_i, e_j)$ on c_j. We call this *partial inference*. If partial inference is not allowed, then the connection validity constraints can be simplified to $d_{i,j} = 1$ only if $e_i = s_j$, which linearizes to two constraints $L \cdot d_{i,j} \leq L + s_j - e_i$ and $L \cdot d_{i,j} \geq L - s_j + e_i$.

Constraint-5: transmission throughput. We only allow flow to pass through valid network connections and the flow that passes should not be larger than the connection's maximum transmission throughput. To enforce this constraint, we add $f_{i,j} \leq d_{i,j} \cdot S_{i,j}$ as a constraint into the MILP problem. $S_{i,j}$ is the maximum number of tokens that can be transmitted through the network connection, which can be calculated via profiling and using methods mentioned in Sec. 3.2.

Optimization target. The MILP problem aims to find a model placement that satisfies all constraints and yields the highest max flow for the cluster. This optimization target can be expressed as maximizing the sum of flow from source, i.e. maximizing $\sum_i f_{source,i}$.

MILP solution orchestration. After the MILP solver finds a solution that satisfies all constraints, we can orchestrate it into a model placement plan and construct the graph abstraction of the cluster. For compute node c_i, s_i and e_i give us the model layers c_i should load into its GPU.

3.4 Analyzing and Speeding up MILP

As Table 2 and 3 show, the number of variables and constraints in the MILP problem scales linearly with the number of compute nodes and network connections. For large clusters with more than 40 nodes, it may take hours before the MILP solver gives a reasonably good solution. To expedite the MILP solving process for large clusters, we introduce three optimizations. First, we prune some of the slow network connections in the cluster. Evaluation in Sec. 5.8 shows that this effectively reduces the problem size without sacrificing much performance. Second, we hint the MILP solver with solutions found by heuristic methods. Since the problem has an exponential solution space, the MILP solver can only cover a small portion within a limited solving time budget. Using solutions from heuristic methods as starting points to the MILP problem expedites the optimization process, especially for large clusters. Sec. 5.8 shows the necessity of starting from heuristic solutions for large clusters. Finally, we notice that the max serving throughput of a cluster is always bounded by the sum of compute throughput of all compute nodes averaged by the total number of layers. The MILP solver uses this as an early-stopping criterion and stops when it finds a solution that is very close to this upper bound.

A common approach for speeding up MILP problems is to relax them to a linear program (LP) by relaxing the integer variables to be linear variables and obtaining a valid solution to the original problem via methods such as rounding the resulting linear variables. We remark that this approach is not viable for the MILP problem formulation above for model placement. This is because the resulting solution from the LP cannot be easily converted to a valid solution of the original problem. The variables for model placement (s_i and b^j_i) decide the edge validity variables $d_{i,j}$, which in turn decides the flow variables $f_{i,j}$. Rounding the non-integral values of model placement variables in the relaxed solution may invalidate some or all network connections and thus drastically changing the max flow of the cluster.

4 Helix Runtime

This section discusses the runtime scheduling of requests in Helix. When the coordinator node receives a new request, it runs Helix's request scheduler to assign the request a *per-request pipeline*, which we will introduce in Sec. 4.1. Then the coordinator node sends the request to the first compute node in the pipeline. When a compute node receives a requests, it performs inference on the request using the layers it is responsible for in that pipeline and send the request to the following compute node. Fig. 3 shows an overview of Helix.

4.1 Scheduler Design: Per-Request Pipelines

To infer a request in the cluster, the scheduler needs to assign a *pipeline* for the request. The pipeline contains a sequence of stages, where each stage specifies a compute node and the layers to infer on the compute node. A valid pipeline must infer each layer of the model in correct order when running the stages sequentially. In Helix, instead of having a group of fixed pipelines and assigning requests to them as previous works [4, 15], we propose a *per-request pipeline* assignment approach, wherein each request will have its own pipeline. The total number of possible pipelines equals to the number of possible paths from source to sink in the graph abstraction of the cluster. The abundant number of pipelines allows the scheduling to fit the capacity of the compute nodes and network connections better. Our Max-Flow formulation enables us to create the per-request pipelines.

The scheduler in Helix uses the graph abstraction of the cluster with model placement solution from MILP, and its max flow solution to guide request scheduling. The scheduler runs on the coordinator node and uses interleaved weighted round-robin (IWRR) [37] to perform request scheduling. IWRR scheduling takes a list of candidates and their weights as input. When queried for scheduling, it selects a candidate with frequency proportional to its weight. Using IWRR allows us to schedule requests following the max flow of the cluster without creating bursts. Below we present our max-flow-based IWRR scheduling process in detail.

Each node in the cluster (including the coordinator node) is represented by an IWRR instance in Helix's scheduler. Each IWRR instance's candidates contain all the nodes connected by valid network connections from the corresponding node. The weight of each candidate equals the flow over the corresponding network connection in the max flow solution obtained in Sec. 3.3. When a new request arrives, the scheduler starts scheduling from the IWRR instance representing the coordinator node. It queries the IWRR instance and gets a compute node c_1, which will host the first pipeline stage. Since the request has not inferred any layers, the first pipeline stage will infer all layers c_1 holds. For the second pipeline stage, the scheduler will use the IWRR instance representing c_1 to determine the compute node c_2 that hosts the second pipeline stage. Because of partial inference mentioned in Sec. 3.3, the layers held by c_1 and c_2 might partially overlap. Therefore, the scheduler will check the overlap and only infer the request on layers previously not inferred in the second stage. The scheduler repeats this process until we form a valid pipeline. Based on the way we construct the graph abstraction of a cluster in Sec. 3, the scheduler can always build a valid pipeline for the cluster. The example in Fig. 4 shows the scheduling of two requests using Helix's request scheduler.

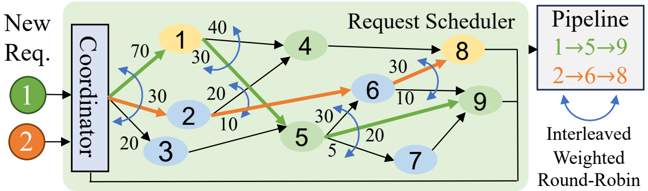


Figure 4: Numbers on each edge represent the flow over the edge in the max flow of the cluster's graph abstraction. Helix's request scheduler uses these numbers as weights to perform interleaved weighted round-robin scheduling to form per-request pipelines. The pipelines used to schedule requests 1 and 2 are shown on the right.

4.2 KV-Cache Estimation

When serving LLMs, each GPU has a limited amount of VRAM to store the KV-cache of requests during inference. If the requests running concurrently on the GPU requires more KV-cache than this limit, the execution engine has to offload some requests to main memory, which significantly harms throughput. Therefore, in the scheduler we maintain an estimation of KV-cache usage of all compute nodes, and mask out compute nodes that exceed high-water mark when running interleaved weighted round-robin. We can schedule more requests to those compute nodes only after some requests currently running on those nodes have finished. This mask ensures that we do not over-subscribe the GPUs' KV-cache.

5 Evaluation

In this section, we aim to answer the following questions.

- Can Helix provide higher throughput compared to existing systems in different settings (Sec. 5.3)?
- Does Helix sacrifice latency to obtain the high throughput (Sec. 5.3)?
- How does network heterogeneity affect Helix's performance (Sec. 5.4)?
- How does the degree of heterogeneity affect Helix's performance (Sec. 5.5)?
- Why does Helix achieve better performance compared to existing systems (Sec. 5.6 and 5.7)?

5.1 Implementation

Helix prototype We implemented a multi-replica pipeline parallel system with 1.5k LoC in Python and 1.7k LoC in C++. For model execution engine, we adopt the latest release of vLLM [16] (0.4.0post1) to avoid reimplementing basic LLM inference optimizations like iteration level scheduling and PageAttention. We implemented a pool of Pages unified for all local layers in a node, since requests may only execute a subset of all local layers. For inter-node communication, we use ZeroMQ [38] to transfer both metadata and intermediate tensors. We use Gurobi [10] as the solver of our MILP.

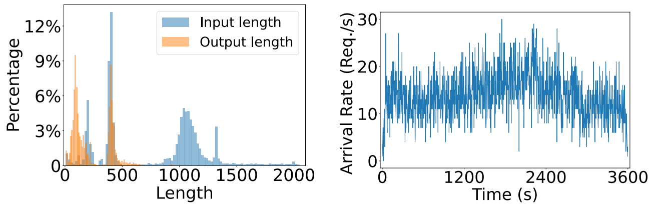
Simulator Besides the prototype system, we build a event-based simulator for distributed LLM inference in heterogeneous clusters with 14k LoC in Python. We tune the simulator with profiled data collected from the prototype system for a high fidelity. Running simulation is much faster and requires less computation than the real system. It also allows us to explore more diverse settings of network conditions, machine heterogeneity and cluster scale. We test the fidelity of the simulator against the prototype system in Sec. 5.3.

5.2 Experiment Setup

Models. We evaluate Helix on LLaMA [40, 41], a representative and popular open-source Transformer model family. Specifically, we use LLaMA 30B and LLaMA 70B to study the system performance on models of different sizes. We run model inference with half-precision (FP16).

Cluster setup. We evaluate with three cluster setups: (1) single cluster, (2) distributed cluster and (3) high GPU heterogeneity. For the *single cluster setup*, we create a heterogeneous cluster with 4 A100 nodes, 8 L4 nodes and 12 T4 nodes within one region on the Google cloud. Our evaluation focuses on single-GPU nodes, as multi-GPU nodes are much harder to allocate on the cloud [39], while our implementation also works for multi-GPU nodes by leveraging tensor model parallelism across GPUs on the same node as supported by vLLM [16]. Average transmission bandwidth between machines is 10 Gb/s and average latency is smaller than 1ms. The *distributed cluster setup* has three clusters that contain 4 A100 nodes, 2 L4 nodes + 8 T4 nodes, and 6 L4 nodes + 4 T4 nodes respectively. Intra-cluster network condition mirrors that of single cluster setup. Inter-cluster communication has an average bandwidth of 100 Mb/s and an average latency of 50 ms (from our profiling results on Google Cloud). For *high GPU heterogeneity* cluster used in simulation, the cluster contains 4 A100 nodes, 6 V100 nodes, 8 L4 nodes, 10 T4 nodes, 4 2×L4 nodes, 6 2×T4 nodes and 4 4×T4 nodes. Multi-GPU nodes run tensor parallelism for the layers assigned to them.

Traces. The traces we use come from Azure Conversation dataset [27]. Fig. 5 shows the length distribution and arrival rate of this dataset. We remove requests with input lengths larger than 2048 or output lengths larger than 1024 to fit in the GPUs we use. The pruned dataset contains 16657 requests with an average input length of 763 and an average output length of 232. We have two settings for arrival rates. For **online setting**, we use real arrival rates from Azure Conversation dataset. We scale the average arrival rate to 75% of the cluster's peak throughput to avoid bursts of requests leading to OOM in the system. For **offline setting**, we allow requests to arrive at the rate needed to fully utilize the cluster. This mimics running offline inference on a dataset. We refer to the two settings as *online* and *offline serving*.



(a) Input and output length.

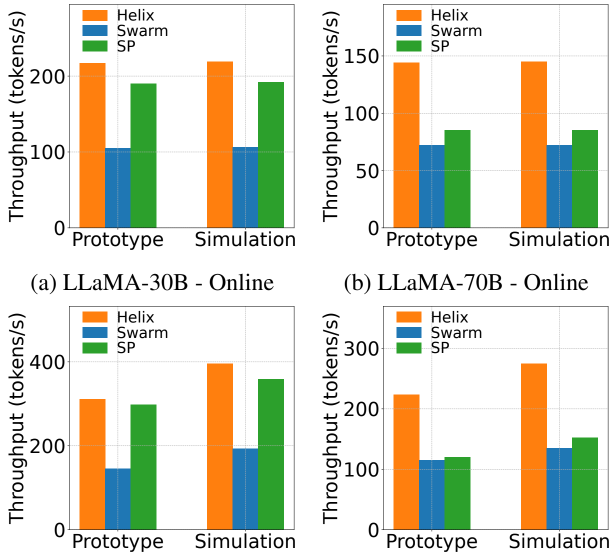
(b) Arrival rate.

Figure 5: Length and arrival rates of requests in Azure Conversation dataset.

Experiment duration. For online setting, we warm up the cluster for 30s and test for 30 minutes. For offline setting, we warm up the cluster for 1 minute and test for 10 minutes. This amount of time is sufficient for our evaluations and we do not run longer to reduce unnecessary experimental cost.

Helix setup. We allow the MILP solver to search w/ and w/o partial inference and w/ and w/o cluster pruning. We also hint the MILP solver with solutions from Petals / Swarm / separate pipelines. Solving times out when the MILP solver does not find better solutions in 10 minutes. The total search budget for each cluster setup is 4 hours on a 14 core CPU.

Baselines. To the best of our knowledge, there are no heterogeneous LLM serving systems with model placement and scheduling applicable to our settings. Therefore, we adopt ideas from a heterogeneous LLM training system, Swarm [31], to build a competitive heterogeneous baseline. We also build another baseline that builds on homogeneous pipelines, which run one separate pipeline for each type of machine. We refer to the two baselines as Swarm and separate pipelines (SP). For Swarm, we implement the method in our system because the original system is designed for training and cannot be directly used for inference. Our implementation ensures that requests in the prompt and decode phase always follow the same path, which is crucial for the correctness of inference. We set the number of pipeline stages to the minimum that allows the weakest GPU in the cluster to hold one stage of layers with half its VRAM. This minimizes the pipeline depth and leaves enough VRAM for KV-cache on GPUs, both of which are crucial to performance. For separate pipelines,



(a) LLaMA-30B - Online (b) LLaMA-70B - Online

(c) LLaMA-30B - Offline (d) LLaMA-70B - Offline

Figure 6: Decode throughput comparison between different methods for online and offline serving in single cluster setup.

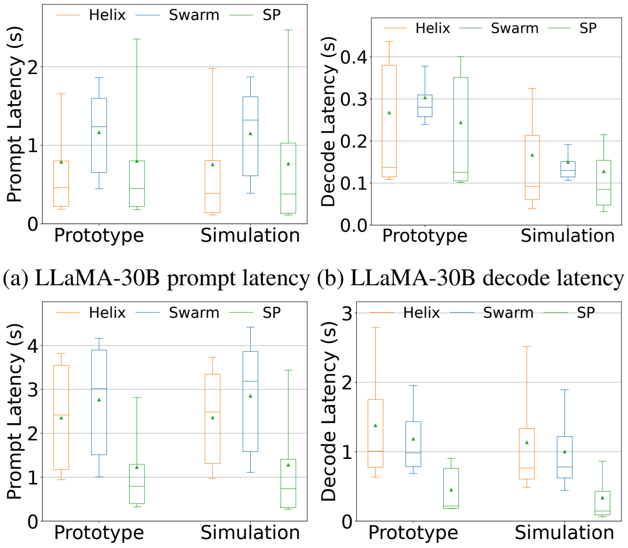
each pipeline serves one replica of the model and layers are equally distributed among machines within the pipeline.

Metrics. For offline serving, we report average *decode throughput*, which is the number of tokens generated per second. For online serving, we further report average *prompt latency* and *decode latency*, which is the average latency for parsing user input and generating new tokens, respectively.

5.3 Single Cluster

Real system evaluation. This section evaluates Helix's prototype system in single cluster setup. We evaluate both LLaMA 30B and LLaMA 70B for online and offline serving, forming four groups of experiments. Each method uses the same model placement plan for online and offline serving. Comparison of decode throughput between different methods for offline and online serving are shown in Fig. 6. Comparison of prompt and decode latency between different methods for online serving is shown in Fig. 7.

Offline serving. For LLaMA 30B, each type of compute node is able to serve a separate pipeline by themselves. In this case, the best model placement found by Helix's model placement planner contains three separate pipelines. The overall performance of Helix is very close to separate pipelines, despite the small performance gain from better utilization of KV cache enabled by the KV cache estimation in Sec. 4.2. Swarm's model placement plan introduces a bottleneck and under-utilizes the A100 nodes, which we will discuss in more detail in Sec. 5.6. As a result, Helix achieves 2.14× boost in decode throughput compared with Swarm.



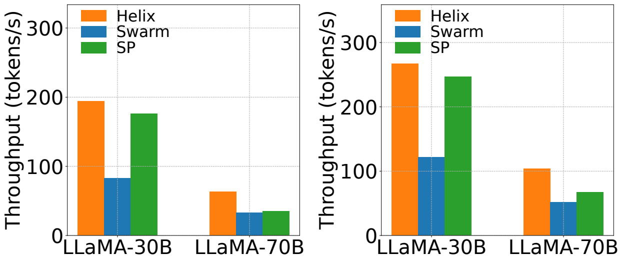
(a) LLaMA-30B prompt latency (b) LLaMA-30B decode latency

(c) LLaMA-70B prompt latency (d) LLaMA-70B decode latency

Figure 7: Prompt and decode latency comparison of online serving in a single cluster setup. In the figure, the box shows the 25 and 75 percentile, the whisker shows the 10 and 90 percentile, the orange line shows the median, and the green triangle shows the mean.

For LLaMA(70B), things are much different. When serving one pipeline with each type of machine, the majority of GPU VRAM will be used for model parameters not leaving sufficient space for the KV cache, which greatly limits the decode throughput. Thus, in this case, the decode throughput of separate pipelines degrades significantly. Similar to LLaMA 30B case, Swarm introduces a bottleneck in its model placement plan, causing under-utilization of both A100 and L4 nodes. As a result, Helix achieves 1.94× and 1.86× decode throughput compared with Swarm and separate pipelines baseline.

Online serving. For LLaMA 30B, Helix has similar prompt and decode latency compared with separate pipelines baseline, since the model placements are identical. On the other hand, Swarm has 47% higher prompt latency and 13% higher decode latency compared with Helix, even though its throughput is just half of Helix (Fig. 6a). The reason is that Swarm under-utilizes the A100 nodes, causing most compute to happen on the slow L4 and T4 GPUs, which will be discussed in detail with an example in Sec. 5.6. For LLaMA 70B, running separate pipelines yields the lowest prompt and decode latency. The reason is that the 8 L4 and 12 T4 nodes are heavily bounded by KV cache capacity (too much VRAM used for parameters) and serves at a very low throughput. The majority of requests are served by the 4 A100 nodes, which has very low latency. This however results in severe waste of compute capacity of the L4 and T4 nodes and thus resulting lower



(a) Online

(b) Offline

Figure 8: Decode throughput comparison for online and offline serving in a distributed cluster setup.

throughput as seen in Fig. 6b.

Simulator fidelity evaluation. We also perform the same experiments in the single-cluster setup on the simulator mentioned in Sec. 5.1. Then we compare its results with that of real machine experiments. Fig. 6a and 6b shows that for online serving experiments, in which the cluster runs at 75% peak throughput, the average decode throughput difference between simulation and real evaluation result is less than 1%. Fig. 7 shows the prompt and decode latency comparison for online serving experiments. We find that for all methods, there is a constant difference of ∼150ms for both LLaMA 30B and 70B. After profiling, we identify this difference as the CPU-GPU data transfer overhead not modeled in the simulator. Since the difference is consistent for all methods, it does not affect the comparison between them. For offline serving experiments, Fig. 6c and 6d shows that the simulator reports a consistently higher throughput for all methods. This gap is also caused by the constant CPU-GPU transfer overhead not modeled. Since the difference is consistent between different methods, the simulator still is valuable as it serves the purpose of comparing the performance of different methods.

5.4 Distributed Clusters

In this section, we evaluate Helix with a distributed cluster setup in simulation. Similar to single cluster experiments, we evaluate on both LLaMA 30B and 70B for online and offline serving. Fig. 8 shows the decode throughput comparison. For LLaMA 30B, the decode throughput of all methods drop by 10-20% compared with single cluster settings. For LLaMA 70B, decode throughput of all methods drop by half. This comes from the slow inter-cluster network connections, and larger models are more affected by this because they require a deeper serving pipeline. For LLaMA 70B, which is heavily affected by the slow network, we find that the max pipeline depth of Helix's model placement is 28% shorter than that of Swarm's, because Helix's MILP model placement planners finds better placement strategies to load more layers into L4 and A100 machines to reduce network communication in this case. By comparing the model placement of Helix for LLaMA 70B in single and distributed cluster setups (which only differs in network condition), the model placement for distributed cluster reduces pipeline depth by 19%. This demonstrates that Helix's model placement takes network condition into account, which makes a 2.0× and 1.5× improvement over Swarm and separate pipelines.

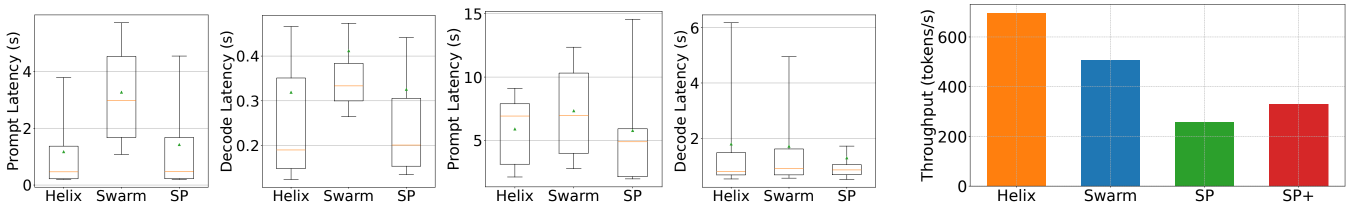
Fig. 9 shows the prompt and decode latency comparison for online serving. For LLaMA 30b, Helix has a similar latency as separate pipelines since both methods use the same model placement in this case. Swarm, on the other hand, has a much higher latency for the same reason as Sec. 5.3. For LLaMA 70B, separate pipelines has the lowest latency, because the 12 slow T4 nodes are bounded by memory and contribute to only a quarter of the total throughput. The under-utilization of T4's compute power improves decode latency by 28% at the cost of 45% decode throughput. For Swarm, we observe severe congestion when the cluster is running at peak throughput. The average prompt latency reaches 70s, which is 7.5× that of Helix's. We will discuss more about the congestion of Swarm with a case study in Sec. 5.7.

5.5 Cluster Heterogeneity

This section evaluates Helix, under increased cluster heterogeneity, with offline serving of LLaMA 70B in a cluster of 42 nodes and 7 node types in simulation. Fig. 9e shows the decode throughput comparison. In this cluster, V100, T4, and T4×2 nodes can not form a separate pipeline by themselves. We report the throughput without these machines for separate pipelines baseline. We also try to build a mixed pipeline using those machines and report the number with the mixed pipeline as separate pipeline+. Results show that Helix achieves 1.38×, 2.72×, and 2.11× throughput compared with Swarm, separate pipelines and separate pipelines+.

5.6 Model Placement Deep Dive

In this section, we analyze the impact of different model placement methods on the serving throughput of the cluster. We test with LLaMA 70B in both single and distributed cluster setups using offline serving. We compare Helix's model placement planner with that of Swarm's and Petals' [4], which is a decentralized heterogeneous LLM inference system. We compare with Petals here because it only has a model placement strategy with no centralized scheduling method and thus not directly applicable in end-to-end evaluation. Swarm divides the model layers into equal length stages and assigns nodes to stages so that each stage has equal amount of compute. Petals, on the other hand, allows nodes to carry different number of layers. It targets on decentralized LLM serving, and thus decides model placement for each node sequentially and greedily selects a placement that covers layers with the least compute. To avoid the impact of different scheduling methods



(a) LLaMA-30B (b) LLaMA-30B (c) LLaMA-70B (d) LLaMA-70B (e) Throughput with more heterogeneity

Figure 9: (a-d) Prompt and decode latency comparison for online serving in the distributed cluster setup. (e) Decode throughput comparison for serving LLaMA 70B in a cluster with 42 nodes and 7 node types.

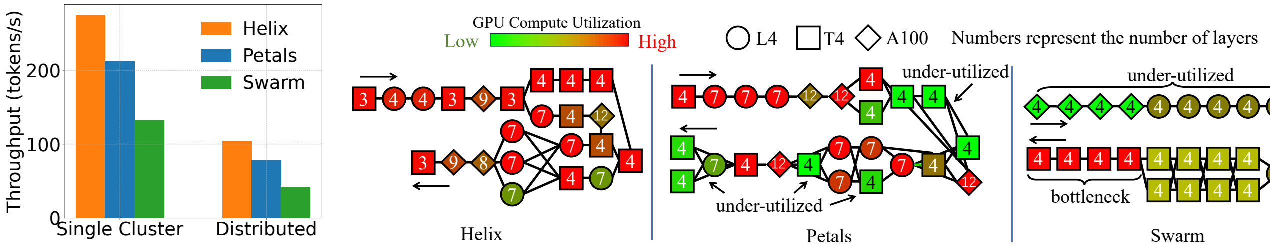


Figure 10: Left: decode throughput of Helix, Petals and Swarm in single and distributed cluster setup. Right: A case study of the model placement found by the three methods for serving LLaMA 70B in single cluster setup. The number in each node represents the number of layers the node holds.

so as to isolate the effect of model placement, we use Helix's request scheduler for all methods. Results in Fig. 10 left show that Helix achieves 1.23× and 2.10× decode throughput against Petals and Swarm respectively in the single cluster setup. This matches our observation in the prototype system. For distributed clusters setup, Helix's relative throughput is 1.34× and 2.49× compared with Petals and Swarm respectively. We use a case study to explain why Helix's model placement method achieves the best performance.

Case study: LLaMA 70B - Single Cluster. Fig. 10 right shows the GPU compute utilization rate of each method's model placement for serving LLaMA 70B in the single cluster setup. The model placement found by Swarm has a bottleneck at the end of its pipeline, where 4 T4 nodes each serves 4 layers. This bottleneck causes GPU under-utilization for A100 and L4 nodes, significantly decreasing the serving throughput. For the model placement of Petals, there is no apparent bottlenecks. However, the placement still under-utilizes 8 T4 nodes and 1 L4 node, negatively affecting the serving throughput. For the placement found by Helix, almost all nodes are full-utilized. The efficient use of compute nodes enables Helix to outperform the baselines by up to 2.10×.

5.7 Request Scheduling Deep Dive

In this section, we analyze the impact of different request scheduling methods on the serving throughput of the cluster. We test with LLaMA 70B in both single and distributed cluster setups using offline serving. We compare Helix's request scheduler with (1) Swarm, which chooses next level nodes with frequency proportional to their throughput, and (2) random scheduling, which randomly chooses a next level node in scheduling. To eliminate the impact of model placement, all methods use the model placement plan found by Helix. Results in Fig. 11 left shows that for single cluster setup, using Helix's scheduling increases serving throughput by 23%. For distributed cluster setup, Helix's throughput increase is 12%. The improvements for single cluster setup match our observation in the prototype system. Moreover, runtime monitoring shows that Swarm and random scheduling have severe congestion. We will further illustrate this in a case study.

Case study: LLaMA 70B - Distributed Clusters. Fig. 11 right shows the model placement plan from Helix's model placement planner for LLaMA 70B in the distributed clusters setup. The model Plan avoids using the slow inter-cluster network connections as much as possible, but there are still a few compute nodes connected with low-bandwidth connections. When using Swarm or random scheduling to schedule requests, we observe severe congestion on the three links marked as "congestion" in the figure – prompt phase requests queue up on those links for an average of 5s - 16s before they can be transmitted. We root-cause the nodes responsible for the congestion and mark them orange in the figure. Surprisingly, we find that one congestion is caused by bad scheduling from a node 3 nodes away. This verifies the necessity of a global scheduling method that can take both network and compute into account. We also observe similar congestion when running Swarm's request scheduling on the model placement

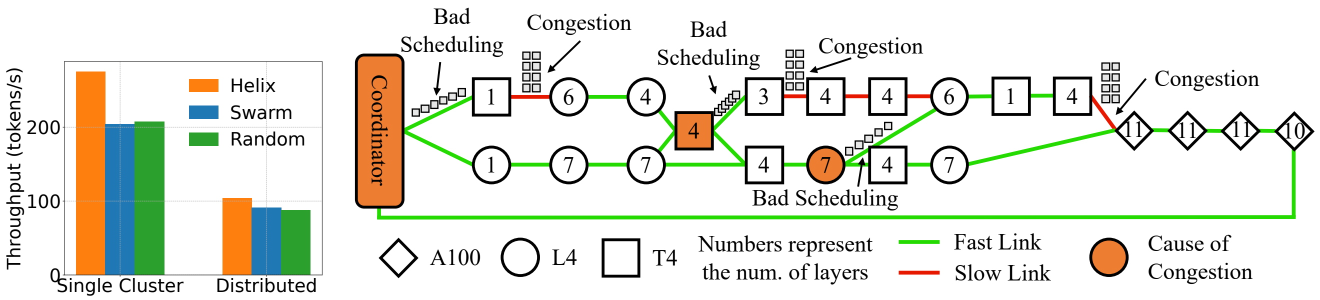
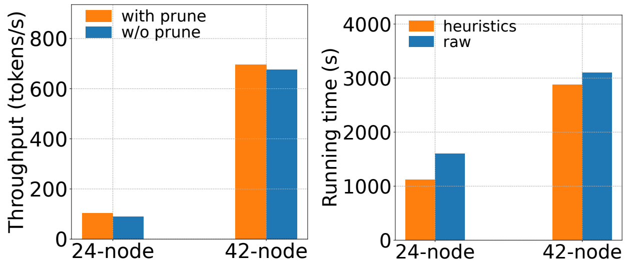


Figure 11: Left: decode throughput of Helix, Swarm and Random scheduling in single and distributed cluster setup. Right: a case study that illustrates the congestion in Swarm and Random scheduling. The model placement used is found by Helix.

Table 4: Problem size with and without pruning. var means variables, and const means constraints.

Problem size	Without pruning	With pruning
24-node	876 var 1122 const	1376 var 1848 const
42-node	2144 var 2772 const	4004 var 5502 const



(a) Cluster pruning

(b) Initial values.

Figure 12: Ablation study.

it finds for LLaMA 70B in the distributed cluster setup.

5.8 Ablation Study on Optimization

This section performs ablation study for the two MILP optimizations introduced in Sec. 3.4. We use the LLaMA 70B settings in Sec. 5.4 and 5.5 to perform the ablation. We refer to them as 24-node and 42-node setting.

Cluster pruning. When cluster pruning is enabled, we prune network connections such that the average degree of each node is 12. This number of connections is sufficient for LLM inference systems as we discuss below. This removes 50% network links for 24-node setting and 72% network links for 42-node setting. Table. 4 shows that pruning reduces problem size by around 40% and 50% for the two settings respectively. We also compare the offline decode throughput of the best model placement found with and without cluster pruning. Results in Fig. 12a shows that Helix achieves 16% and 4% better performance when we enable pruning in the two settings respectively, for this problem instance. We note that the amount of speed-up achieved would vary depending on the specific instance of the MILP problem at hand. The reason is that in LLM serving, network connections used is very sparse – usually each node only communicates with a few other nodes. Also, there are many equivalent model placements that can achieve the same throughput. Pruning the cluster very likely keeps some of these placements still valid. It makes the search for these placements easier with limited optimization time, as the problem size (and solution space size) is reduced.

Initial values. We compare the performance of running Helix's model placement planner starting from solutions of heuristic methods and from default values. Since the best model placement found is the same for both methods, we compare the wall clock time to find the placement. Fig. 12b shows that running MILP from heuristic solutions takes 43% and 8% more time in the two setups respectively, for this problem instance. We note that the amount of speed-up achieved would vary depending on the specific instance of the MILP problem at hand. The results demonstrate that start from heuristic solutions expedites model placement in Helix.

6 Related Work

Machine Learning Model Serving There are a large number of works for serving machine learning models, discussing aspects including system implementation [23, 24], model placement [18, 28, 34, 43], and request scheduling [9, 34, 48]. However, due to LLM's unique auto-regressive execution paradigm, these approaches fail to efficiently serve LLMs. Instead, many recent LLM-specific systems tackle the unpredictable execution time and high memory consumption in LLM serving. Orca [46] proposed iteration level scheduling to release resources once a request is finished. vLLM [16] introduced PageAttention to further reduce the memory consumption of each request by allocating exact number of pages it requires. Speculative Inference [17, 22] applies a small model to predict multiple output tokens, and verify them in a single iteration. Splitwise [27] and DistServe [50] found that disaggregating the prompt and decode phase can improve the throughput, since the two phases have different workload characteristics. Sarathi [2] introduced chunked prefill, which allocates a budget to the prompt phase to make each microbatch's workload balanced, minimizing pipeline bubble. All above works are orthogonal to our work and can be integrated into our system, since our focus is on the cluster heterogeneity.

ML Workloads on Heterogeneous Cluster Several methods explored the potential of utilizing heterogeneous GPUs for ML tasks. Some of them [14, 26] co-design the model partition and placement on a heterogeneous cluster but assume a uniform network bandwidth. Learninghome [32] and DeDLOC [7] studied the network-aware routing on a decentralized cluster but only considers either data or pipeline parallelism individually. SWARM [31], as discussed in Sec. 2.1, optimized the pipeline communication in a heterogeneous network. However, it schedules only by the next stage's metadata, lacking a global view. There are also several efforts on using approximations to reduce the network communication [42] or synchronization [12]. Most of them focus on model training. In model inference, especially LLMs, serving with heterogeneous and geo-distributed GPUs is not well studied. SkyPilot [45] and Mélange [8] selects the best type of GPUs for a request, but each request is served by a single GPU type. Petals [4], as discussed in Sec. 2.1, studies a decentralized pipeline parallel setup. It designs a greedy model allocation and request scheduling for a dynamical device group, losing optimizing opportunities for a fixed device group.

7 Conclusion

This paper presents Helix, the first high-throughput, low-latency LLM serving engine for heterogeneous GPU clusters. Helix formulates and solves layer placement and request scheduling as a Max-Flow problem. Compared to existing solutions, Helix achieves significant improvements in throughput and latency.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

[3] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

[4] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Max Ryabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[6] Joseph Cheriyan and SN Maheshwari. Analysis of preflow push algorithms for maximum network flow. *SIAM Journal on Computing*, 18(6):1057–1086, 1989.

[7] Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Anton Sinitsin, Dmitry Popov, Dmitry V Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, et al. Distributed deep learning in open collaborations. *Advances in Neural Information Processing Systems*, 34:7879–7897, 2021.

[8] Tyler Griggs, Xiaoxuan Liu, Jiaxiang Yu, Doyoung Kim, Wei-Lin Chiang, Alvin Cheung, and Ion Stoica. M\'elange: Cost efficient large language model serving by exploiting gpu heterogeneity. *arXiv preprint arXiv:2404.14527*, 2024.

[9] 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.

[10] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[11] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[12] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia:{Geo-Distributed} machine learning approaching {LAN} speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 629–647, 2017.

[13] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[14] Xianyan Jia, Le Jiang, Ang Wang, Wencong Xiao, Ziji Shi, Jie Zhang, Xinyuan Li, Langshi Chen, Yong Li, Zhen Zheng, et al. Whale: Efficient giant model training over heterogeneous {GPUs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 673–688, 2022.

[15] Youhe Jiang, Ran Yan, Xiaozhe Yao, Beidi Chen, and Binhang Yuan. Hexgen: Generative inference of foundation model over heterogeneous decentralized environment. *arXiv preprint arXiv:2311.11514*, 2023.

[16] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[17] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[18] 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.

[19] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

[20] Meta. Introducing Meta Llama 3: The most capable openly available LLM to date — ai.meta.com. https://ai.meta.com/blog/meta-llama-3/. [Accessed 07-05-2024].

[21] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

[22] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

[23] NVIDIA. Triton Inference Server.

[24] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[25] OpenAI. ChatGPT, 2023.

[26] Jay H Park, Gyeongchan Yun, M Yi Chang, Nguyen T Nguyen, Seungmin Lee, Jaesik Choi, Sam H Noh, and Young-ri Choi. {HetPipe}: Enabling large {DNN} training on (whimpy) heterogeneous {GPU} clusters through integration of pipelined model parallelism and data parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 307–321, 2020.

[27] Pratyush Patel, Esha Choukse, Chaojie Zhang, Íñigo Goiri, Aashaka Shah, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. *arXiv preprint arXiv:2311.18677*, 2023.

[28] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems*, 5, 2023.

[29] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[30] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[31] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient. In *International Conference on Machine Learning*, pages 29416–29440. PMLR, 2023.

[32] Max Ryabinin and Anton Gusev. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. *Advances in Neural Information Processing Systems*, 33:3659–3672, 2020.

[33] Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

[34] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: A gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[35] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[36] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is cross-region the answer? In *Proceedings of the 4th Workshop on Machine Learning and Systems*, pages 107–116, 2024.

[37] Seyed Mohammadhossein Tabatabaee, Jean-Yves Le Boudec, and Marc Boyer. Interleaved weighted round-robin: A network calculus analysis. *IEICE Transactions on Communications*, 104(12):1479–1493, 2021.

[38] The ZeroMQ authors. ZeroMQ An open-source universal messaging library, 2024.

[39] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[41] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[42] Jue Wang, Yucheng Lu, Binhang Yuan, Beidi Chen, Percy Liang, Christopher De Sa, Christopher Re, and Ce Zhang. Cocktailsgd: Fine-tuning foundation models over 500mbps networks. In *International Conference on Machine Learning*, pages 36058–36076. PMLR, 2023.

[43] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent {GPU} sharing in container clouds for deep learning workloads. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 69–85, 2023.

[44] x.ai. Annocing grok.

[45] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, et al. {SkyPilot}: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, 2023.

[46] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, 2022.

[47] Binhang Yuan, Yongjun He, Jared Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy S Liang, Christopher Re, and Ce Zhang. Decentralized training of foundation models in heterogeneous environments. *Advances in Neural Information Processing Systems*, 35:25464–25477, 2022.

[48] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. {MArk}: Exploiting cloud services for {Cost-Effective},{SLO-Aware} machine learning inference serving. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1049–1062, 2019.

[49] Shiwei Zhang, Lansong Diao, Chuan Wu, Zongyan Cao, Siyu Wang, and Wei Lin. Hap: Spmd dnn training on heterogeneous gpu clusters with automated program synthesis. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 524–541, 2024.

[50] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving. *arXiv preprint arXiv:2401.09670*, 2024.