Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow

Yixuan Mei
yixuanm@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Yonghao Zhuang
yzhuang2@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Xupeng Miao
xupeng@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Juncheng Yang
juncheny@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Zhihao Jia
zhihao@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Rashmi Vinayak
rvinayak@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

This paper introduces Helix, a distributed system for high-throughput, low-latency large language model (LLM) serving in heterogeneous GPU clusters. The key idea behind Helix is to formulate inference computation of LLMs over heterogeneous GPUs and network connections as a *max-flow* problem on directed, weighted graphs, 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 on heterogeneous GPUs. 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 clusters ranging from 24 to 42 GPU nodes shows that Helix improves serving throughput by up to 3.3× and reduces prompting and decoding latency by up to 66% and 24%, respectively, compared to existing approaches. Helix is available at https://github.com/Thesys-lab/Helix-ASPLOS25.

CCS Concepts: • **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → *Artificial intelligence*; *Parallel computing methodologies.*

Keywords: large language model serving, system for ML, distributed systems, cloud computing

ACM Reference Format:
Yixuan Mei, Yonghao Zhuang, Xupeng Miao, Juncheng Yang, Zhihao Jia, and Rashmi Vinayak. 2025. Helix: Serving Large Language Models over Heterogeneous GPUs and Network via Max-Flow. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3669940.3707215

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707215

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-2 [56]	70 billion	12	7	4
GPT-3 [1]	175 billion	30	18	9
Grok-1 [59]	314 billion	53	32	16
LLaMA-3 [10]	405 billion	68	41	21

Table 2. Availability of different GPU instances in 6 regions on Google Compute Engine [11].

Region	GPU Type					
	H100	A100 80GB	A100 40GB	L4	T4	V100
us-central-1	✓	✓	✓	✓	✓	✓
us-east-4	✓	✓	×	✓	✓	×
us-east-1	×	×	✓	✓	✓	✓
eu-west-3	✓	×	×	✓	✓	×
asia-ne-1	✓	×	✓	✓	✓	×
asia-ne-3	×	×	✓	✓	✓	×

1 Introduction

Generative large language models (LLMs) such as GPT-4 [1] and LLaMA-3 [28] have demonstrated exceptional capabilities of creating natural language texts across a spectrum of application domains, including chatbot [38], coding assistant [26, 44], and task automation [15]. 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 [61] and vLLM [22]) target *homogeneous* GPU clusters [29], where all GPUs are of the same type and have identical memory capacity and compute resources. Due to increasing model

Table 3. Properties of GPUs deployed in today's data centers. We report SXM version for H100 and A100. Data collected from NVIDIA data sheet [32–35].

GPU	FP16 (TFLOPs)	Memory (GB)	Bandwidth (GB/s)	Power (W)	Price (USD)
H100 [33]	1979	80	3350	700	25k ~40k
A100 [32]	312	40	1555	400	10k ~15k
L4 [34]	242	24	300	72	~3k
T4 [35]	65	16	300	70	~1k

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. Recent works have identified that it is increasingly difficult to allocate GPUs of this magnitude within a single cloud region [50, 60].

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. Table 2 illustrates the *heterogeneous* GPU deployment in Google Compute Engine [11], where datacenters are equipped with various NVIDIA GPUs including H100, A100, V100, L4, and T4. These heterogeneous GPU instances are spread across 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. As Table 3 shows, eight NVIDIA L4 GPUs can offer comparable FP16 compute performance to a single NVIDIA H100 GPU, while providing greater memory capacity, lower power consumption, and a more cost-effective price point. Moreover, the availability of these GPUs vary significantly across regions. We empirically find that obtaining higher quotas and securing GPU instances is much easier in some regions than others on Google Cloud Engine, and the availability of different GPUs vary a lot (see Table 4).

Geo-distributed LLM serving with heterogeneous GPUs enables the aggregation of available GPUs from multiple regions. This approach not only enhances resource utilization but also minimizes LLM serving costs by strategically leveraging the most cost-effective GPU instances across various geographical locations. Similarly, there is also a trend of using volunteer consumer GPUs to address the GPU scarcity problem [9, 46, 62]. 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 computation over heterogeneous devices or geo-distributed environments. However, prior attempts either focus on long-running training workloads [16, 27, 39, 45, 64], which cannot adapt to LLM serving scenarios with real-time inference requests, or focus on decentralized serving with volunteer computing [5, 6], which lack the global coordination necessary to efficiently use GPU and network resources in clusters.

Table 4. GPU quotas and deployment limits in us-east and asia-southeast during our evaluation of Helix. We measure deployment limits at two distinct time periods.

Region	Quota			Max Deployed		
	A100 40GB	L4	T4	A100 40GB	L4	T4
us-east	8	8	16	0	0	16
asia-southeast	8	24	32	4	12	>20

To efficiently serve LLMs over heterogeneous GPUs and network, we propose Helix, a distributed system for high-throughput, low-latency LLM serving. Helix's key idea is to formulate the execution of LLM serving over heterogeneous GPUs and network as a *data flow* problem under the constraints of diverse GPU computing capabilities, memory capacities, as well as complex inter-GPU connections. Helix leverages mixed integer linear programming to determine optimal model placement under these constraints. To accommodate heterogeneity, Helix introduces *per-request pipelines*, where each request has its own independent pipeline for scheduling. This combination of flow-based formulation and per-request pipelines enables Helix to achieve high GPU utilization in heterogeneous and geo-distributed GPU clusters. We will discuss the challenges and Helix's solutions in Sec. 3.

We have implemented Helix on top of vLLM [22] 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-1 30B and LLaMA-2 70B. Compared to heterogeneity-aware baselines, Helix improves serving throughput by up to 3.3× while reducing average prompting and decoding latency by up to 66% and 24%.

In summary, our contributions are:

- A system for LLM serving in heterogeneous and geo-distributed GPU clusters.
- A max-flow formulation for LLM serving and an MILP-based algorithm to optimize model placement.
- Flexible flow-based per-request pipelines to maximize GPU utilization.
- An implementation of our techniques and an evaluation on various LLM benchmarks.

2 Background

2.1 LLM Architecture and Serving

Most of today's LLMs adopt a decoder-only Transformer architecture [7, 42], 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, known as the *prompt phase*, the model processes all prompt tokens and generates the first output token. In subsequent iterations, 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 (⟨eos⟩). 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 [61] 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 [22] introduces PagedAttention, managing memory for KV-cache with identical pages and allocating a new page only when a request has used up all its pages; multi-query [47] and group-query attention [3] modifies the self-attention mechanism to reduce the size of KV-cache stored for each token.

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) [49] 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) [17] 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, and 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 devices. As LLMs grow in size and the latest generation GPUs remain scarce, deploying these models across heterogeneous computing devices has become a critical necessity, a challenge that previous research has not adequately addressed.

Deploying LLMs across geo-distributed, heterogeneous GPU clusters requires careful consideration of both hardware capabilities and network characteristics. Simple equal distribution of model layers across devices fails to maximize the potential of more powerful hardware. Moreover, the significant bandwidth differences between intra- and inter-regional network connections must inform both model placement and request scheduling decisions, particularly when infrastructure spans multiple geographical regions.

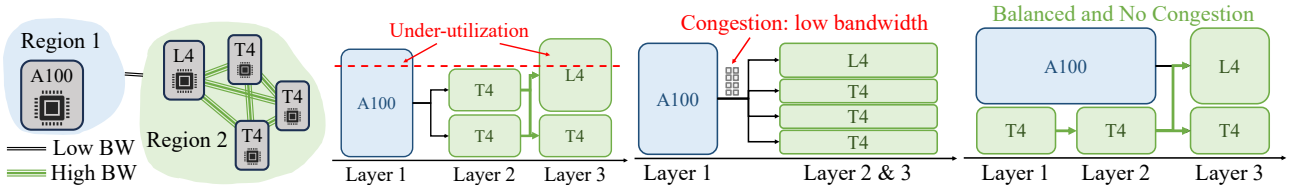
No prior work has focused on LLM serving on heterogeneous GPU clusters. The most related work is Petals [5], which performs decentralized LLM serving with volunteer computing. Users contribute GPUs to form a decentralized swarm and newly joined machines greedily serve the pipeline stages with least compute capacity. For request scheduling, users greedily choose the server with lowest latency to them. Petals is effective for volunteer computing, but the lack of global coordination makes it unable to fully utilize the GPUs and network when the cluster is a known priori. Another relevant work is SWARM [45], which performs DNN training in heterogeneous clusters. It evenly partitions the model into pipeline stages. When routing a request to the next pipeline stage, it selects the replica based on real-time throughput of candidates. Our evaluation shows that such simple heuristics are not enough to achieve good performance in geo-distributed clusters with heterogeneous GPUs and network.

3 Opportunities and Challenges

Using heterogeneous and geo-distributed GPUs presents new opportunities for LLM serving. As shown in Table 3, multiple commodity GPUs (L4, T4) can match the compute capacity of high-end GPUs (H100, A100) while offering advantages in memory capacity, energy efficiency, and cost. Furthermore, Tables 4 and 7 demonstrate that GPU availability varies significantly across regions, yet inter-region network conditions remain suitable for LLM serving, motivating a geo-distributed approach. However, leveraging heterogeneous and geo-distributed GPUs poses several key challenges.

3.1 Challenge 1: Model Placement

Due to the increasing size of LLMs, serving them on modern GPUs requires employing *tensor* [49] and *pipeline* [17, 39]



(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; 1b) Model placement by uniformly partition the model, then allocate devices by a balanced compute capacity; 1c) Co-optimizing model partition and device placement to make the compute capacity more balanced; 1d) Co-optimizing model partition, device placement, and request scheduling in a network-aware way.

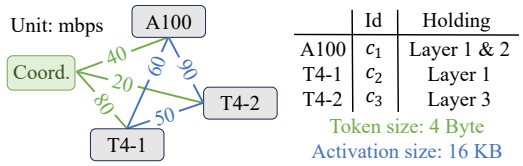
model parallelism to partition an LLM into stages and place those stages on different GPUs, a task we term *model placement*. Homogeneous serving systems (e.g., Orca [61]) partition an LLM into equal-sized stages and assign them to GPUs. This approach results in sub-optimal utilization of high-performance GPUs as it accommodates the memory and computational limitations of less powerful GPUs. Existing heterogeneity-aware serving systems (e.g., Petals [5]) 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.

Helix's solution: Helix exploits the flexibility of token-level scheduling in LLM serving and 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 [5, 45]. Leveraging the data dependencies and homogeneity of LLM layers, Helix expresses the MILP problem with linear number of variables and constraints relative to the number of compute nodes and network connections, resulting in a tractable problem size.

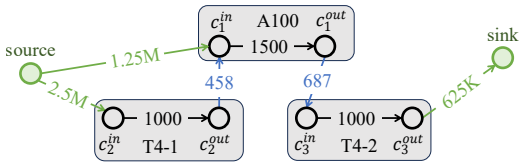
3.2 Challenge 2: Request Scheduling

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.

Helix's solution: 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.



(a) A 3-node cluster with model placement. Network connections between the coordinator and compute nodes transmit tokens (4 Byte) while others 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.

4 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.

4.1 Formulation of LLM Serving

A cluster to serve LLMs generally contains one coordinator node h and a group of compute nodes C. Each compute node $c_i \in C$ has a compute capacity and GPU VRAM size. Compute nodes with multiple GPUs can be abstracted as a single logical node, aggregating GPUs' combined computational capacity and GPU VRAM resources. Throughput and latency of network connections between nodes in the cluster are also given. Based on the cluster information, LLM serving requires finding a placement of model layers to compute nodes to maximize a *serving performance metric* that will be defined below. The model placement function $\Psi : C \mapsto \mathcal{P}(\mathcal{M})$ takes

as input a compute node and returns a (usually continuous) subset of the model $\mathcal{M}$. Here, we assume pipeline parallelism for inter-node parallelization and tensor parallelism for intra-node parallelization across GPUs, as tensor parallelism demands extensive communication and may be constrained by network. One widely used metric [5, 45] to assess a model placement is the performance of the pipeline stage with the lowest compute capacity $\min_i \sum_j \text{capacity}(j) \cdot 1_{i \in \Psi(j)}$, where $\text{capacity}(j)$ is the compute capacity of the j^{th} node. 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.

4.2 Necessity of Joint Optimization

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 as a Max-Flow problem. In this example, the clusters 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 follow the common approach, which statically partitions the model and then assigns 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 run in a data parallel manner for the rest of the layers. However, communications from one pipeline stage to another become a 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 divides the workload between the A100 GPU and the two T4 GPUs, reducing communication on the slow link.

4.3 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/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 the current model placement. The following shows the formal construction.

Table 5. 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

For a given cluster with coordinator node h, a set of compute nodes 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 and coordinator nodes. For each compute node $c_i \in C$, 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 in 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. For the coordinator node, we represent it as source and sink vertices in the graph.

Network connections. In a given cluster, a node may communicate with any other nodes, creating $O(|C|^2)$ 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 the connection bandwidth divided by the transmission size of an activation (tens of kilobytes). Helix performs a one-time profiling and uses the average bandwidth as the connection bandwidth. The capacity of the edges models the throughput constraint imposed by the speed of network connection between different nodes. We denote the full set of possible network connections by $\mathcal{E}$.

Table 6. Constraints used in MILP.

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

After constructing the equivalent graph abstraction of a cluster, we run the preflow-push algorithm [8] 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 network connection in one second. Therefore, the max flow gives us the max possible serving throughput of the cluster with current model placement.

4.4 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 a linear number of variables and constraints with respect to the number of compute nodes and network connections. The 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 at most three linear constraints with the help of at most two auxiliary variables. An overview of the variables and constraints is shown in Table 5 and 6.

Node variables. To represent the model placement on each compute node, we introduce two groups of variables in our MILP formulation. Suppose that 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 integer 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. 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. 4.3). 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 similar to 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 $0 \le s_i < L$ and $e_i \le L$. (c_i holds layers in range $[s_i, e_i)$)

Constraint-2: flow conservation. For each compute node c_i, the sum of flow that goes in to c_i^{in} must be 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 enumerate 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 not exceed its maximum inference throughput. We can impose this constraint with $\sum_u f_{ui} \le \sum_{j=1}^{k} b_i^j \cdot T_j$. 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 if requests can be transmitted through them. For a network connection from the coordinator node to the 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: $s_i \le L \cdot (1 - d_{source,i})$. Similarly, for network connection from compute node c_i to coordinator, we constrain its validity with $L \cdot d_{i,sink} \le e_i$. For network connection from compute node c_i to c_j, its validity $d_{i,j}$ is determined by whether $s_j \le e_i < e_j$ 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 $s_j \le e_i$, which can be linearized as $(L+1)(1 - cond_{i,j}^1) \ge s_j - e_i$. $cond_{i,j}^2$ takes value 1 only if $e_i < e_j$, which can be linearized as $e_j - e_i \ge 1 - (L+1)(1 - cond_{i,j}^2)$. The network connection is valid only if both binary auxiliary variables are true, which can be expressed as $d_{i,j} \le 0.5 * cond_{i,j}^1 + 0.5 * cond_{i,j}^2$.

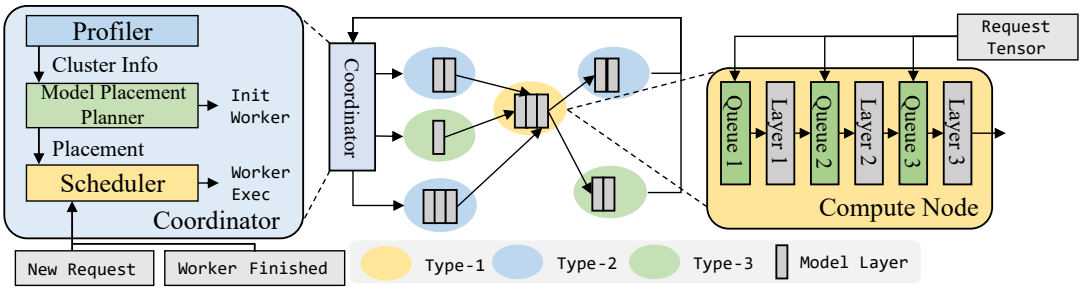


Figure 3. Helix overview. In Helix, the coordinator plans model placement as described in Sec. 4.4. We only need to run model placement once for each cluster. When a new request arrives, the coordinator node runs Helix scheduler to assign it a per-request pipeline and sends it to the first node in the 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.

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 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. 4.3.

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.

4.5 Analyzing and Speeding up MILP

As Table 5 and 6 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 still 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. 6.8 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 for the MILP problem expedites the optimization process, especially for large clusters. Sec. 6.8 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 stop criterion and stops when it finds a solution that is very close to this upper bound. We remark that, for further scaling of Helix to hundreds or even thousands of nodes, one viable approach is to first partition the nodes into multiple smaller clusters using heuristics and then apply Helix independently.

4.6 Replacing MILP with LP or Heuristics?

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 above. 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_i^j) 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.

Our preliminary exploration also indicates that using simpler heuristics cannot guarantee good performance across various cluster setups, because of the exponential solution space of this problem. Sec. 6.6 shows that the model placement found by Helix with MILP is much better than that of the heuristic baselines.

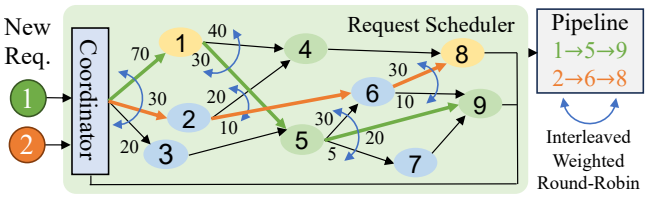
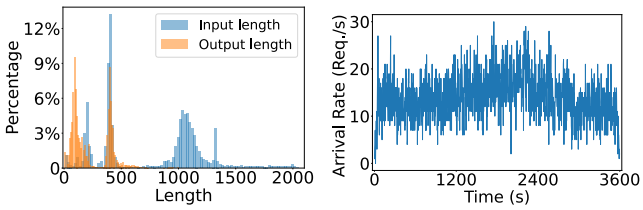


Figure 4. Topology graph of a cluster, where each vertex is a compute node, and each edge is a valid network connection. Numbers over edges represent the flow over the network connection in the max flow solution. The pipelines used to schedule requests 1 and 2 are shown on the right.



(a) Length distribution. **(b)** Arrival rate.

Figure 5. Statistics of Azure Conversation dataset.

5 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. 5.1. 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 sends the request to the following compute node. Fig. 3 shows the overview of Helix.

5.1 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 exactly-once and in correct order when running the stages sequentially. Existing works [5, 19] use fixed pipelines, in which each pipeline contains a disjoint set of machines, and assign requests to those pipelines. In Helix, instead of using fixed-pipelines, we propose a *per-request pipeline* assignment approach, wherein each request will have its own pipeline and the pipelines may intersect with each other. The total number of possible pipelines equals 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 better fit the capacity of the compute nodes and network connections. Our Max-Flow formulation enables us to create the per-request pipelines.

The Helix request scheduler performs scheduling based on the cluster's *topology graph* (Fig. 4). In the topology graph, vertices correspond to the nodes in the cluster. Directed edges correspond to the valid network connections (under the model placement found by solving the MILP in Sec. 4). We bind an interleaved weighted round-robin (IWRR) [51] scheduler to each vertex. The IWRR scheduler takes as input a list of candidates and their weights. To schedule a request, it selects a candidate with frequency proportional to its weight. For each vertex u, the IWRR scheduler's candidates contain all vertices v such that directed edge (u, v) exists (i.e. (u, v) represents a valid network connection). The weight of v equals the flow over the network connection of (u, v) in the max flow solution. Using IWRR allows us to schedule requests following the max flow without creating bursts.

Helix's request scheduler runs on the coordinator node. When a new request arrives, it first uses the IWRR scheduler of the vertex representing the coordinator node to determine the compute node c_1 for the first pipeline stage. Then, it uses the IWRR scheduler of the vertex representing c_1 to determine the compute node c_2 for the second stage, and repeats this process until a valid pipeline is established. Fig. 4 shows the scheduling of two requests. After setting the request's pipeline, the coordinator node sends the request to the first compute node in the pipeline to begin inference. During inference, Helix adopts a *dynamic batching* strategy where each node includes all requests received during the processing of the previous batch to form a new one. This best-effort batching occurs without additional waiting periods.

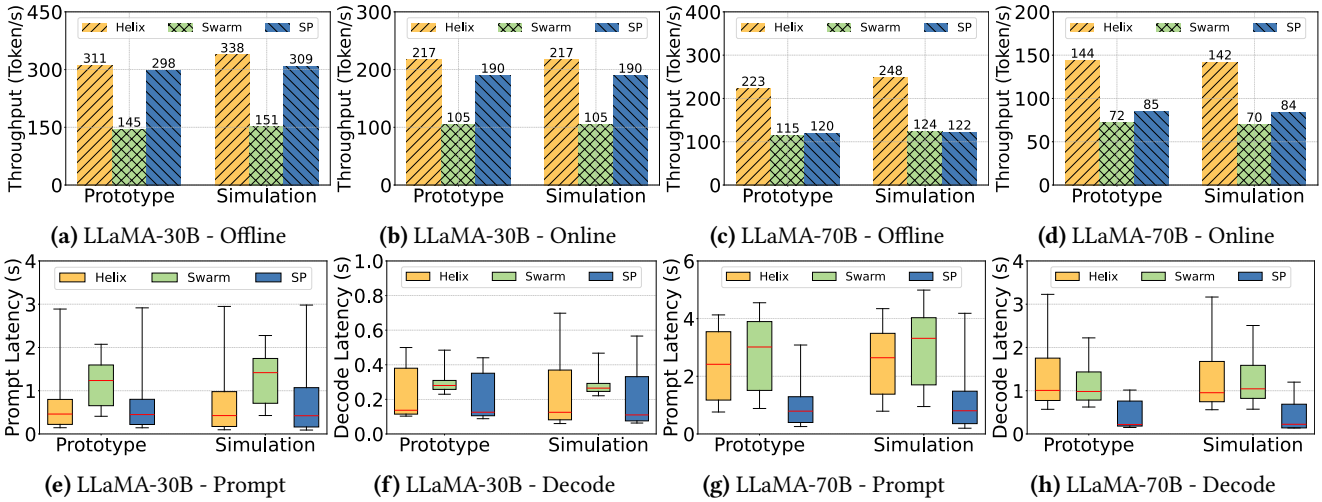
5.2 KV-Cache Estimation

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 require more KV-cache than this limit, the execution engine has to offload some requests to main memory, which significantly harms throughput. However, we do not know exactly how much KV-cache each request will use because the length of the output is unknown before inference finishes. Therefore, in the scheduler we maintain an estimation of KV-cache usage of all compute nodes using average output length, and mask out compute nodes that exceed the high water mark when running IWRR. We can schedule more requests to the compute nodes only after some requests currently running on those nodes have finished. This mask ensures that we do not oversubscribe the GPU's KV-cache.

6 Evaluation

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

- Can Helix provide higher throughput for heterogeneous GPUs in a single cluster? (Sec. 6.3)
- Does Helix sacrifice latency for throughput? (Sec. 6.3)
- Can Helix provide higher throughput for heterogeneous GPUs in geo-distributed clusters? (Sec. 6.4)



(a) LLaMA-30B - Offline **(b)** LLaMA-30B - Online **(c)** LLaMA-70B - Offline **(d)** LLaMA-70B - Online

(e) LLaMA-30B - Prompt **(f)** LLaMA-30B - Decode **(g)** LLaMA-70B - Prompt **(h)** LLaMA-70B - Decode

Figure 6. Single cluster results: (a - d) decode throughput, **(e - h)** prompt and decode latency for online serving. The box shows the 25 and 75 percentile. The whisker shows the 5 and 95 percentile. The red line shows median.

Table 7. Network bandwidth between machines in asia-east, us-central, europe-west and australia-southeast. Measured with iperf3 on Google Compute Engine.

Receiver	Sender			
	asia-east2-a	us-central1-f	eu-west3-c	au-se1-c
asia-east2-a	/	123 Mbps	67 Mbps	175 Mbps
us-central1-f	122 Mbps	/	204 Mbps	123 Mbps
eu-west3-c	61 Mbps	196 Mbps	/	54 Mbps
au-se1-c	159 Mbps	118 Mbps	63 Mbps	/

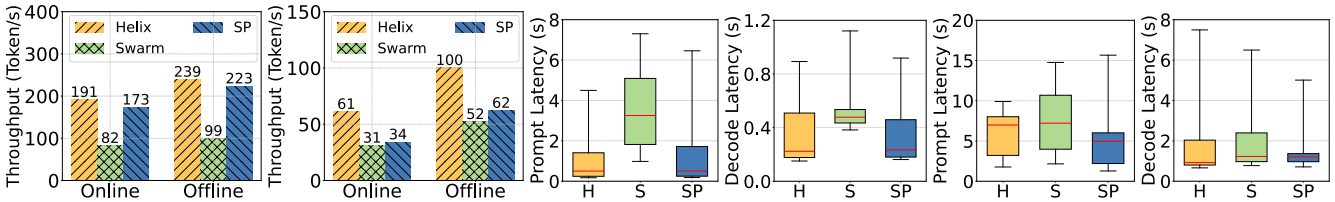
- Can Helix provide consistently high throughput when the degree of GPU heterogeneity increases? (Sec. 6.5)
- Why does Helix achieve better performance compared to existing systems (Sec. 6.6 and 6.7)?

6.1 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, we adopt the latest release of vLLM [22] (0.4.0post1) to avoid re-implementing basic LLM inference optimizations. We implemented a *unified page pool* atop vLLM to support partial inference. We use ZeroMQ [52] for inter-node communication and Gurobi [14] as MILP solver.
Simulator. We also implemented a simulator for distributed LLM inference with 14k LoC in Python. It supports simulation for heterogeneous GPUs and network conditions. It gives us the flexibility to explore more diverse settings for network, GPU heterogeneity and cluster scale. Sec. 6.3 evaluates the fidelity of the simulator and shows that the simulation errors are less than 5% for all metrics.

6.2 Experiment Setup

Models. We evaluate Helix on LLaMA [55, 56], a representative and popular open-source Transformer model family. Specifically, we use LLaMA-1 30B and LLaMA-2 70B to study the system performance on models of different sizes. We run model inference with half-precision (FP16). In subsequent sections, we refer to these two models as LLaMA 30B and LLaMA 70B.
Cluster setup. We evaluate with three cluster setups: (1) single cluster (Sec. 6.3), (2) geo-distributed clusters (Sec. 6.4), and (3) high GPU heterogeneity cluster (Sec. 6.5). First, the *single cluster* setup contains 4 A100 nodes, 8 L4 nodes and 12 T4 nodes connected by 10Gb/s network. We configure the network bandwidth to 10 Gb/s, as this rate is sufficient to ensure that LLM serving is limited by GPU throughput rather than network capacity. We allocate the nodes within one region on the Google Cloud. Second, the *geo-distributed clusters* contain three clusters with (i) 4 A100 nodes, (ii) 2 L4 nodes + 8 T4 nodes, and (iii) 6 L4 nodes + 4 T4 nodes. Inter-cluster communication has an average bandwidth of 100 Mb/s and an average latency of 50 ms. We configure the bandwidth to 100 Mb/s to simulate cross-region network limitations, based on our profiling results in Table 7. Finally, the *high GPU heterogeneity 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. The latter two setups are performed in simulation. Our evaluation of the prototype system focuses on single-GPU nodes, as multi-GPU nodes are much more difficult to allocate in the cloud [53]. Our implementation also works for multi-GPU nodes by leveraging tensor model parallelism across GPUs on the same node as supported by vLLM [22].
Traces. The traces we use come from Azure Conversation dataset [40]. 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 maintain reasonable runtime memory usage for vLLM.



(a) LLaMA-30B **(b)** LLaMA-70B **(c)** LLaMA-30B **(d)** LLaMA-30B **(e)** LLaMA-70B **(f)** LLaMA-70B

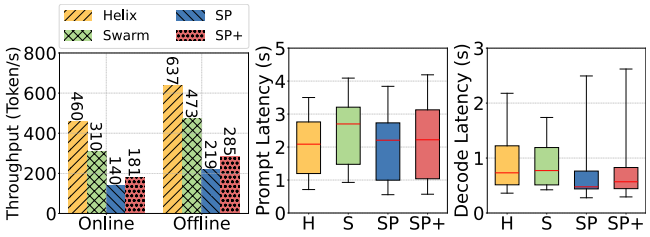
Figure 7. Geo-distributed clusters: (a - b) decode throughput, **(c - f)** prompt and decode latency for online serving. "H" stands for Helix, "S" stands for Swarm. Box: 25 and 75 percentile. Whisker: 5 and 95 percentile. Red line: median.

The pruned dataset contains 16657 requests with an average input length of 763 and an average output length of 232. We remark that this dataset is more challenging than the ones used by prior work [22] due to longer request length. 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*.

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 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 max 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 [45], to build a competitive heterogeneous baseline. We also build another baseline that handles GPU heterogeneity by serving one model replica with each type of machine. We refer to the two baselines as Swarm and separate pipelines (SP). To ensure a fair comparison, we implemented the SP baseline within our system rather than utilizing other systems. We further compare with the model placement of Petals [5] in Sec. 6.6, which is a decentralized LLM serving system for volunteer computing. We do not compare with Petals in end-to-end serving as it lacks centralized request scheduling. For Swarm, we implemented their model placement and scheduling algorithm in our system, since their original system can not be used for inference. We set the number of pipeline stages to the minimum that allows the weakest GPU to hold one stage with half its VRAM. This minimizes the pipeline depth and leaves enough VRAM for KV-cache, both of which are crucial to performance.



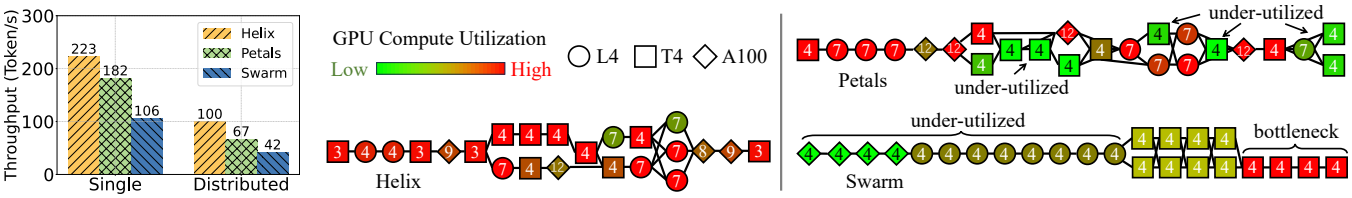
(a) LLaMA-70B **(b)** LLaMA-70B **(c)** LLaMA-70B

Figure 8. High GPU-heterogeneity clusters setup: (a) decode throughput, **(b - c)** prompt and decode latency for online serving. "H" stands for Helix, "S" stands for Swarm.

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.

6.3 Single Cluster

This section evaluates Helix with online and offline serving of LLaMA 30B and 70B in the single cluster setup. Fig. 6 shows the results. For LLaMA 30B, each GPU type has enough nodes to serve at least one individual pipeline. The best model placement discovered by Helix serves three replicas of the model, each with one type of GPU. As a result, Helix and SP achieve similar performance: Helix has 4% and 14% higher decode throughput for offline and online serving because of its better KV-cache utilization; the prompt latency is 2% lower and decode latency is 10% higher. Compared with Swarm, Helix achieves 2.14× and 2.07× higher decode throughput for offline and online serving. Helix also reduces prompt and decode latency by 32% and 12% for online serving. We find that Swarm's model placement introduces a bottleneck and under-utilizes the A100 nodes, which we discuss in more detail in Sec. 6.6. We note that the comparatively higher latency observed in our experiments, relative to other LLM serving systems [22, 40], can be attributed to our use of less powerful GPUs (T4 and L4) in contrast to the A100 and H100 GPUs utilized in other studies.



(a) Decode throughput. **(b)** Model placement case study. Numbers on each node represent the number of layers the node holds.

Figure 9. (a) Comparing different model placement methods with offline serving of LLaMA 70B. **(b)** The case study shows model placements for serving LLaMA 70B on the single cluster setup.

For LLaMA 70B, nodes from a single GPU type can not serve one model replica by themselves while leaving enough VRAM for KV-cache. In this case, SP's throughput significantly decreases. Compared with SP, Helix achieves 1.86× and 1.69× higher decode throughput in offline and online serving. The average latency of Helix is higher than SP because SP serves majority of requests with A100 nodes and under-utilizes the L4 and T4 nodes. Compared with Swarm, Helix achieves 1.94× and 2.00× higher decode throughput in offline and online serving. Helix's prompt latency is 15% lower, but decode latency is 16% higher because of Helix's high GPU utilization. Similar to LLaMA 30B, Swarm under-utilizes A100 nodes because of its model placement.

We also conduct these experiments in the simulator (see Sec. 6.1) to examine its fidelity. The simulation results are shown alongside the real system evaluation results in Fig. 6. The average error of decode throughput is lower than 5%. The average error of prompt and decode latency is lower than 5% and 4% respectively. This indicates that our simulator achieves high fidelity and serves the purpose of comparing the performance of different methods.

6.4 Geo-Distributed Clusters

This section evaluates Helix for both online and offline serving of LLaMA 30B and 70B in geo-distributed clusters using our high-fidelity simulator in Sec. 6.1. Fig. 7 shows the results. Although this setup involves the same set of GPUs as the single cluster, the slow inter-cluster network causes all methods to have lower throughput and higher latency.

For LLaMA 30B, the best model placement found by Helix still consists of three pipelines served by three types of machines separately. Compared with SP, Helix has 7% and 10% higher decode throughput for offline and online serving. The prompt latency is 14% lower and decode latency is 2% higher. Compared with Swarm, Helix achieves 2.41× and 2.33× higher decode throughput for both online and offline serving. Helix also reduces prompt and decode latency by 66% and 24% for online serving.

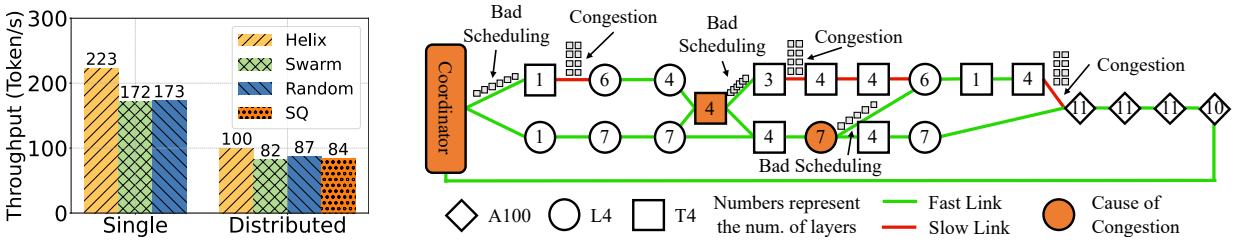
On the other hand, serving LLaMA 70B is more sensitive to slow network because of larger activation size and model depth. Helix reduces network overhead by using model placements with shallower pipeline depth. The model placement found by Helix reduces pipeline depth by 28% compared to Swarm. It is also 19% shallower than the one found by Helix when network is fast. Helix's model placement planner balances network overhead with single node's GPU utilization, achieving high throughput and low latency in geo-distributed GPU clusters. Compared with SP, Helix achieves 1.61× and 1.79× higher decode throughput for offline and online serving, respectively. SP has lower average latency because it serves most requests with fast GPUs, while 12 T4 nodes are under-utilized because of VRAM constraints. Compared with Swarm, Helix achieves 1.92× and 1.97× higher decode throughput for offline and online serving. Helix also reduces prompt and decode latency by 21% and 7%. We observe severe congestion during offline serving with Swarm. The average prompt latency reaches 71s, which is 7.5× that of Helix's. We will discuss more about the congestion of Swarm with a case study in Sec. 6.7.

6.5 High GPU-Heterogeneity Cluster

This section evaluates Helix on a highly heterogeneous GPU cluster with 42 compute nodes and 7 types of GPUs. Fig. 8 shows the results. In this cluster, V100, T4, and T4×2 nodes cannot form serving pipelines by themselves. We report the throughput without those machines for SP. We also try to build a mixed pipeline using those machines and report the number with the mixed pipeline as SP+. Compared with Swarm, SP and SP+, Helix achieves 1.37×, 2.91×, and 2.24× throughput for offline serving, and 1.48×, 3.29× and 2.54× for online serving. Helix also reduces prompt latency by 17%, 1% and 7% respectively. The average decode latency of Helix is slightly higher, because the baselines under-utilize the slow GPUs and serve most requests with fast GPUs with large VRAM. This result indicates that Helix can achieve consistently high performance when there are many types of GPUs in the cluster.

6.6 Model Placement Deep Dive

In this section, we analyze the impact of different model placement methods on the serving throughput of the cluster. We evaluate Helix's offline serving performance on both single and geo-distributed clusters, and compare Helix's model placement method with those used in Swarm and Petals.



(a) Decode throughput. **(b)** Request scheduling case study.

Figure 10. (a) Comparing different request scheduling methods with offline serving of LLaMA 70B. **(b)** A case study that illustrates the congestion in Swarm and random scheduling when serving LLaMA 70B in geo-distributed clusters setup.

Both methods perform model placement using throughput-based heuristics. To isolate the effect of model placement, we use Helix's request scheduler for all methods. Fig. 9a shows the decode throughput. Compared with Petals and Swarm, Helix achieves 1.23× and 2.10× higher throughput on the single cluster, and 1.49× and 2.38× throughput on the geo-distributed clusters. We perform a case study on LLaMA 70B to demonstrate why Helix achieves the best performance.

Case study: LLaMA 70B - single cluster. Fig. 9b shows the model placement and GPU compute utilization for each method when serving LLaMA 70B on the single cluster. Swarm's model placement introduces a bottleneck at the end of its pipeline, where 4 T4 nodes each serves 4 layers. This bottleneck causes GPU under-utilization on A100 and L4 nodes, significantly decreasing the serving throughput. For Petals' model placement, 8 T4 nodes and 1 L4 node are under-utilized, which negatively affects the serving throughput. For Helix's model placement, almost all nodes are fully-utilized. The efficient use of GPUs enables Helix to outperform Swarm and Petals by 2.10× and 1.23× respectively.

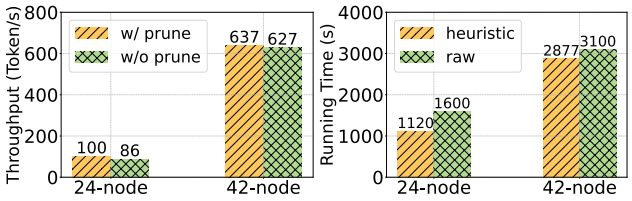
6.7 Request Scheduling Deep Dive

This section analyzes the impact of different request scheduling methods on the serving throughput of the cluster. We evaluate Helix's offline serving performance of LLaMA 70B on both the single cluster and geo-distributed clusters. We compare Helix's request scheduler with (1) Swarm, which schedules requests based on real-time throughput of each candidate node, and (2) random scheduling, which randomly chooses a candidate in scheduling. For the geo-distributed clusters setup, we further compare with Shortest Queue First scheduling (SQ), which always assigns requests to the node with shortest queue. To eliminate the impact of model placement, all methods use the model placement found by Helix. Fig. 10a shows the decode throughput. Compared with Swarm and random scheduling, Helix achieves 30% and 29% higher throughput on the single cluster, and 22% and 15% higher throughput on the geo-distributed clusters. Compared with Shortest Queue First scheduling, Helix achieves 19% higher decode throughput. Moreover, runtime monitoring shows that all three baseline scheduling methods introduce severe congestion. We illustrate this further in the case study.

Case study: LLaMA 70B - distributed clusters. Fig. 10b shows the model placement plan found by Helix for serving LLaMA 70B on the geo-distributed clusters. The plan avoids using slow inter-cluster network connections as much as possible, but a few compute nodes are still 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 hops 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 serving LLaMA 70B with Swarm's request scheduling on the model placement it finds for the geo-distributed clusters.

Table 8. Problem size with and without pruning. var means variables, and cstr means constraints.

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



(a) Cluster pruning. **(b)** Initial values.

Figure 11. Ablation study on MILP optimization.

6.8 Ablation Study on Optimization

This section performs an ablation study on the two MILP optimizations introduced in Sec. 4.5. We evaluate offline serving of LLaMA 70B on the geo-distributed and highly

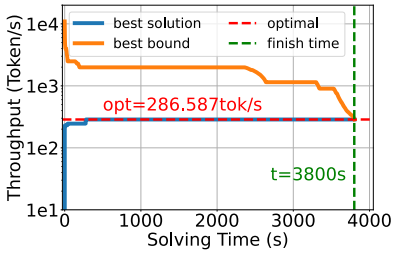


Figure 12. Best solution and best upper bound found by the MILP solver relative to solving time. The red dotted line marks the optimal throughput for this cluster.

heterogeneous clusters, referred to as the 24-node and 42-node settings respectively.

Cluster pruning. When cluster pruning is enabled, we prune network connections such that the average degree of each node is 12, which is sufficient for LLM inference systems as we discuss below. Enabling cluster pruning removes 50% and 72% network connections for 24 and 42-node settings. Table. 8 shows that pruning reduces problem size by 36% and 46% for the two settings. Fig. 11a shows that Helix achieves 16% and 2% higher decode throughput when using the model placement found with cluster pruning. We note that the amount of speed-up achieved would vary depending on the specific instance of the MILP problem at hand. Pruning slow network connections does not harm throughput because network connections used in serving 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 placements found are the same, we compare the wall clock time to find the placement. Fig. 11b shows that running MILP from heuristic solutions takes 43% and 8% less time for the 24- and 42-node setup. We note that the speed-up achieved would vary depending on the specific instance of the MILP problem at hand. The results show that starting from heuristic solutions accelerates model placement in Helix.

6.9 Model Placement Quality

This section evaluates the quality of model placements found by Helix relative to the MILP solving time. In this evaluation, we run Helix to find the optimal model placement for serving LLaMA 30B in a cluster with 4 L4 and 6 T4 machines. We record the best model placement as well as the best upper bound found by the MILP solver that we used (Gurobi) during the solving process. The best upper bound represents the best possible objective value that could be achieved for the MILP problem and will gradually become tighter as the solver explores more nodes and adds cutting planes. Fig. 12 shows the quality of the best model placement and the best upper bound relative to the solving time. Results show that Helix finds the optimal solution in less than 5 minutes, but it takes the solver more than one hour to enumerate all possible solutions and confirm the optimality. This indicates that we can early-stop the solving process, as high-quality solutions emerge in the early stages of computation.

7 Related Work

Machine Learning Model Serving There are a large number of works for serving machine learning models, discussing aspects including system implementation [36, 37], model placement [24, 41, 48, 58], request scheduling [13, 48, 63], and tail-latency mitigation [20, 21, 31]. 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 [61] proposed iteration level scheduling to release resources once a request is finished. vLLM [22] introduced PageAttention to further reduce the memory consumption of each request by allocating exact number of pages it requires. Speculative Inference [23, 30] applies a small model to predict multiple output tokens, and verify them in a single iteration. Splitwise [40] and DistServe [65] 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 clusters. Several methods have utilized heterogeneous GPUs for ML tasks. Some of them [18, 39] co-design the model partition and placement on a heterogeneous cluster but assume a uniform network bandwidth. Learninghome [46] and DeDLOC [9] studied the network-aware routing on a decentralized cluster but only considers either data or pipeline parallelism individually. SWARM [45], 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 network communication [57] or synchronization [16]. Most of them focus on model training. In model inference, especially LLMs, serving with heterogeneous and geo-distributed GPUs is not well studied. SkyPilot [60] and Mélange [12] select the best type of GPUs for a request, but

each request is served by a single GPU type. Petals [5], 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. HexGen [19] is a concurrent work on LLM serving in heterogeneous clusters. It is based on fixed pipelines (see Sec. 5.1) and adopts heuristic-based methods to search for the optimal model placement. In comparison, our Max Flow formulation and per-request pipeline is more flexible than HexGen and our MILP formulation can guarantee the optimal solution.

Scheduling for Heterogeneous Resources There exists extensive research on scheduling algorithms for heterogeneous resources. For example, energy-aware scheduling [25] in the Linux kernel schedules tasks for heterogeneous CPU topologies. There are also several works [4, 43, 54] that focus on general scheduling in heterogeneous clusters. Due to the auto-regressive nature of LLM inference, these methods cannot be directly used for heterogeneous LLM serving.

8 Conclusion

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

Acknowledgment

We thank the anonymous reviewers and our shepherd Íñigo Goiri for their valuable feedback and constructive suggestions that helped improve this paper. We also express our gratitude to the Google Cloud Innovator program for providing the machines on Google Compute Engine for our experiments. This work was supported in part by a Sloan Foundation Fellowship and a VMware Systems Research Award. This work was also partially supported by the National Science Foundation under grant numbers CNS-2147909, CNS-2211882, and CNS-2239351, along with gift awards from Amazon and Meta.

A Artifact

A.1 Abstract

Our artifacts include Helix's simulator and prototype system for distributed LLM serving in heterogeneous and geo-distributed clusters. The code implements key algorithms including the MaxFlow-based LLM serving formulation, MILP-based model placement planner, and per-request pipeline scheduler. We also provide comprehensive documentation and scripts for environment setup and system execution from scratch. The artifacts contain the identical simulator and prototype system used in Helix's evaluation in the paper. The simulator can run on a single machine, while the prototype system requires a cluster deployment.

A.2 Artifact check-list (meta-information)

- **Algorithm:** MaxFlow-based LLM serving formulation; MILP-based model placement planner; per-request pipeline scheduler
- **Compilation:** The simulator is implemented in Python. The prototype system's inter-node communication framework is written in C++ (recommended to compile with GCC 13.2) and includes an automated compilation script with CMake, while its remaining components are implemented in Python.
- **Model:** We use LLaMa-2 70B as the test workload. The prototype system operates with dummy weights, requiring only the model architecture specification (which is provided in our code repository).
- **Data set:** We use the Azure Conversation Dataset as our evaluation traces, with a pre-parsed version included in our code repository.
- **Run-time environment:** For the simulator, we recommend using Python 3.10, while it can also support other recent Python versions. The simulator is not sensitive to OS versions. For the prototype system, we recommend using Ubuntu 24.04 LTS to setup the environment. We also recommend using conda to isolate the run-time environment for both systems. For detailed software dependencies, please refer to Sec. A.3.3.
- **Hardware:** The simulator runs on a single machine without specific hardware requirements. The prototype system runs on a cluster of 24 machines with NVIDIA GPUs. Please refer to Sec. A.3.2 for detailed hardware requirements.
- **Metrics:** We evaluate using the same metrics as presented in the paper: decoding throughput, decoding latency, and prompt processing latency.
- **Output:** Results are logged to both terminal output and files. We include expected results in the code repository.
- **How much disk space required (approximately)?:** The total size of all log files is around 300 MB.
- **How much time is needed to prepare workflow (approximately)?:** The environment setup takes around 1 - 2 hours.
- **How much time is needed to complete experiments (approximately)?:** Functionality evaluation takes around 2 hours. Reproducibility evaluation takes around 16 hours. (The part that needs the whole 24 machine cluster is around 4 hours. **Namely, the sections that need the whole cluster are part of Sec 6.3, 6.6 and 6.7.**)
- **Publicly available?:** Yes. DOI: 10.5281/zenodo.14037926.
- **Code licenses (if publicly available)?:** Apache 2.0

A.3 Description

A.3.1 How to access. The artifact is publicly available at https://github.com/Thesys-lab/Helix-ASPLOS25. It is also archived as 10.5281/zenodo.14037926. The file size is around 300 MB. Please refer to our Github repository for the latest version.

A.3.2 Hardware dependencies. The simulator runs on a single machine without specific hardware requirements. We recommend at least 32 GB of memory for simulating large

clusters to prevent out-of-memory issues. The prototype system requires cluster deployment - our example configuration uses 24 machines, consisting of 4 machines with 1×A100-40GB, 8 machines with 1×L4, and 12 machines with 1×T4. We recommend to set up each machine with at least 16 CPU cores and 128 GB memory to avoid out-of-memory issues. The network connection between machines should ideally be at least 10 Gbps, with latency of approximately a few milliseconds. Usually machines in the same region from common cloud providers can meet the network requirements. For functionality test purposes, it is also possible to use machines from different regions.

A.3.3 Software dependencies. We recommend running the simulator with Python 3.10. It relies on networkx, matplotlib and gurobipy. The MILP-based model placement planner uses Gurobi as the MILP solver. Running the example in the code base does not require additional licenses. However, if you want to run model placement for larger clusters, it is necessary to acquire a Gurobi license that does not limit problem size. We recommend running the prototype system on Ubuntu 24.04 LTS and with Python 3.10. To build the inter-node communication framework, you need to install build-essential, cmake, libzmq, cppzmp and pybind11. To run the prototype system, you need to install CUDA 12.6 and vLLM 0.4.0.post1. For a step-by-step guide to setting up the environment and running the experiments, please refer to https://github.com/Thesys-lab/Helix-ASPLOS25/blob/master/readme.md.

A.3.4 Data sets. We use the Azure Conversation Dataset as our evaluation traces, with a pre-parsed version included in our code repository.

A.3.5 Models. We use LLaMa-2 70B as the test workload. The prototype system operates with dummy weights, requiring only the model architecture specification.

A.4 Detailed Steps for Reproducing Results

We provide an example to show the functionality of our system, please refer to https://github.com/Thesys-lab/Helix-ASPLOS25/blob/master/readme.md. We also provide another example to reproduce all result we got in the paper, please refer to https://github.com/Thesys-lab/Helix-ASPLOS25/blob/master/artifact_evaluation/ae_readme.md

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] Rashmi Bajaj and Dharma P Agrawal. Improving scheduling of tasks in a heterogeneous environment. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):107–118, 2004.

[5] 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.

[6] Alexander Borzunov, Max Ryabinin, Artem Chumachenko, Dmitry Baranchuk, Tim Dettmers, Younes Belkada, Pavel Samygin, and Colin A Raffel. Distributed inference and fine-tuning of large language models over the internet. *Advances in Neural Information Processing Systems*, 36, 2024.

[7] 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.

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

[9] 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.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[11] Google Cloud. Google cloud compute products. https://cloud.google.com/products/compute, 2024. Accessed: 2024-03-13.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Youhe Jiang, Ran Yan, Xiaozhe Yao, Yang Zhou, Beidi Chen, and Binhang Yuan. Hexgen: Generative inference of large language model over

heterogeneous environment. In *Forty-first International Conference on Machine Learning*, 2024.

[20] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 30–46, 2019.

[21] Jack Kosaian, KV Rashmi, and Shivaram Venkataraman. Learning-based coded computation. *IEEE Journal on Selected Areas in Information Theory*, 1(1):227–236, 2020.

[22] 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.

[23] 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.

[24] 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.

[25] Linux Kernel Documentation. Energy aware scheduling. https://docs.kernel.org/scheduler/sched-energy.html, 2024. Linux Kernel Documentation.

[26] 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.

[27] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Realhf: Optimized rlhf training for large language models through parameter reallocation, 2024.

[28] 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].

[29] 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.

[30] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 932–949, 2024.

[31] Hema Venkata Krishna Giri Narra, Zhifeng Lin, Ganesh Ananthanarayanan, Salman Avestimehr, and Murali Annavaram. Collage inference: Using coded redundancy for lowering latency variation in distributed image classification systems. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 453–463. IEEE, 2020.

[32] NVIDIA. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet-us-nvidia-1758950-r4-web.pdf. Accessed: 2024-10-28.

[33] NVIDIA. NVIDIA H100 Tensor Core GPU. https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet. Accessed: 2024-10-28.

[34] NVIDIA. NVIDIA L4 Tensor Core GPU. https://resources.nvidia.com/en-us-data-center-overview/l4-gpu-datasheet. Accessed: 2024-10-28.

[35] NVIDIA. NVIDIA T4 Tensor Core GPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-datasheet-951643.pdf. Accessed: 2024-10-28.

[36] NVIDIA. Triton Inference Server.

[37] 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.

[38] OpenAI. ChatGPT, 2023.

[39] 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.

[40] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pages 118–132. IEEE, 2024.

[41] 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.

[42] 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.

[43] Andrei Radulescu and Arjan JC Van Gemund. Fast and effective task scheduling in heterogeneous systems. In *Proceedings 9th heterogeneous computing workshop (HCW 2000)(Cat. No. PR00556)*, pages 229–238. IEEE, 2000.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

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

[53] 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.

[54] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems*, 13(3):260–274, 2002.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] x.ai. Annocing grok.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 193–210, 2024.