HEXGEN: Generative Inference of Large Language Model over Heterogeneous Environment

Youhe Jiang [*1] **Ran Yan** [*1] **Xiaozhe Yao** [*2] **Yang Zhou** [3] **Beidi Chen** [3] **Binhang Yuan** [1]

Abstract

Serving generative inference of the large language model is a crucial component of contemporary AI applications. This paper focuses on deploying such services in a heterogeneous and cross-datacenter setting to mitigate the substantial inference costs typically associated with a single centralized datacenter. Towards this end, we propose HEXGEN, a *flexible distributed inference engine* that uniquely supports the asymmetric partition of generative inference computations over both tensor model parallelism and pipeline parallelism and allows for effective deployment across diverse GPUs interconnected by a fully heterogeneous network. We further propose a *sophisticated scheduling algorithm* grounded in constrained optimization that can adaptively assign asymmetric inference computation across the GPUs to fulfill inference requests while maintaining acceptable latency levels. We conduct an extensive evaluation to verify the efficiency of HEXGEN by serving the state-of-the-art LLAMA-2 (70B) model. The results suggest that HEXGEN can choose to achieve up to $2.3\times$ lower latency deadlines or tolerate up to $4\times$ more request rates compared with the homogeneous baseline given the same budget. Our implementation is available at `https://github.com/Relaxed-System-Lab/HexGen`.

1. Introduction

Large language models are distinguished by the vast scale of parameters being trained over a substantial pre-train corpus. Such extensive training enables them to be remarkably adaptable across a broad spectrum of downstream tasks (Bommasani et al., 2021). In fact, large language models such as GPT-4 (Bubeck et al., 2023), Llama2-70B (Touvron et al., 2023), and Falcon-180B (Institute, 2023) have essentially revolutionized the way AI systems are developed and deployed, which have nourished a large number of advanced applications. In such an ecosystem, serving the generative inference requests for large language models presents a critical challenge — given the unprecedented model scale, unlike classic machine learning models, parallel inference strategies have to be leveraged to accommodate the high computational and memory demands while ensuring low-latency generative inference outcomes.

The state-of-the-art inference service of the large language model is usually hosted in a single centralized data center with homogeneous high-performance GPUs, which can be very expensive in terms of the cloud service fee. The high cost of such deployment potentially limits the democratization of this great technique. Alternatively, the deployment of the large language model inference over a heterogeneous cross-datacenter environment can be a promising direction to reduce the inference cost, which has not been fully explored. The heterogeneous environment for foundation model inference service can encompass a wide range of options, including more affordable cloud services (such as spot instances (Thorpe et al., 2023; Athlur et al., 2022) and serverless computing (Guo et al., 2022)) to even fully decentralized platforms (Yuan et al., 2022; Borzunov et al., 2023) that leverage a diverse set of GPUs contributed by volunteers in an extreme setting.

However, deploying large language model inference across a heterogeneous environment presents some unique challenges. Unlike traditional machine learning models, large language model inference consists of two different phases: a prompt phase that handles a sequence of input tokens at once and a decoding phase where output tokens are generated step-by-step. Additionally, large language models require the adoption of specialized *parallel inference strategies* to effectively distribute the intensive computations across multiple GPUs. The two most commonly employed approaches are *tensor model parallelism* and *pipeline parallelism*. In

[*]Equal contribution [1]Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China [2]Department of Computer Science, ETH Zurich, Zürich, Switzerland [3]Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania. Correspondence to: Binhang Yuan <biyuan@ust.hk>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

terms of coordinating such a complicated distributed computation, there are some fundamental challenges stemming from the *heterogeneity*:

- **Heterogeneous GPU computation capacity.** To fully leverage the economic GPU computation power, it is essential to employ a range of GPU types, each has distinct peak FLOPS, GPU device memory bandwidth, and GPU device memory constraints. However, most (if not all) distributed implementations of large language model inference frameworks do *not accommodate this GPU diversity*, as they typically assume a homogeneous GPU cluster configuration. This results in an implementation that enforces complete symmetry in the distribution of the inference computation, e.g., requiring every pipeline stage to adhere to the same tensor model parallel degree, which essentially limits the performances of the distributed inference over heterogeneous GPUs.
- **Heterogeneous GPU connection.** The heterogeneity of the cross-GPU connection is even more significant. In a standard homogeneous setting, the intra-machine GPU connections usually rely on the same NVLink or PCIe, and the inter-machine GPU connections are often based on RDMA. While in a fully heterogeneous setting, the connections between each pair of GPUs can vary significantly, including both fast NVLink or PCIe connections and cross-geo-region slow connections. This heterogeneity of connections forces a scheduling algorithm to consider *exponentially many more distinct allocation strategies* when compared with the homogeneous setting.

In order to overcome these challenges, we propose HEXGEN, a large language model inference system that coordinates distributed inference computation over a set of GPUs with different computation capacities connected by heterogeneous connections. We provide flexible support of asymmetric parallel execution within the scope of tensor model parallelism and pipeline parallelism to accommodate the heterogeneous GPU computation capacity. We also propose a scheduling algorithm to determine an efficient allocation of inference computation under diversified connections. Concretely, we make the following contributions:

Contribution 1. We implement the HEXGEN system with the support of asymmetric tensor model parallelism and pipeline parallelism. Essentially, HEXGEN allows each pipeline parallel stage to be assigned with a different number of layers and tensor model parallel degrees to flexibly accommodate the heterogeneity of different GPUs and fully unleash the potential of heterogeneous GPU powers.

Contribution 2. We formally define the scheduling problem of serving the inference of multiple copies of the same foundation model concurrently over a set of heterogeneous GPU devices as a constrained optimization problem. We concretely define computation cost, communication cost, and memory limits for such inference workflow. We then propose a two-phase optimization solution, where we use a dynamic programming algorithm to define the optimal layer of each pipeline, and a heuristic-based evolutionary algorithm to efficiently search for the optimal layout.

Contribution 3. We evaluate HEXGEN by conducting a comprehensive empirical study to compare the system and economic efficiency between the heterogeneous setting enabled by HEXGEN and the standard homogeneous setting within a centralized data center serving the state-of-the-art LLAMA-2 (70B) model. We show that given the *same* budget in terms of cloud service fees, HEXGEN can choose to achieve up to $2.3\times$ lower latency deadlines or tolerate up to $4\times$ more traffic request rate compared with the homogeneous baseline. Additionally, when given only *half* of the budget, HEXGEN can still maintain a similar level of inference service compared to the homogeneous baseline.

Overview. The rest of the paper is organized as follows. We provide some preliminaries in Section 2; introduce our asymmetric parallel implementation for the generative inference in Section 3 and scheduling algorithm in Section 4; present the experimental results in Section 5, summarize related work in Section 6, and conclude in Section 7.

2. Preliminary

Generative inference computation. A typical generative inference task of the transformer-based foundation model consists of two stages: i) the *prefill* stage that takes a prompt sequence to compute the key-value cache (KV cache) for each transformer layer of the large language model; and ii) the *decoding* stage which utilizes the previous KV cache to generate new tokens step-by-step and appends the new KV cache. The inference computation can be summarized below. Denote the batch size by b, the prompt sequence length by s^{in}, the output sequence length by s^{out}, the hidden dimension of the transformer by H, and the total number of transformer layers by L. Given the weight matrices of a transformer layer specified by $\mathbf{w}_K^\kappa, \mathbf{w}_Q^\kappa, \mathbf{w}_V^\kappa, \mathbf{w}_O^\kappa \in \mathcal{R}^{H\times H}$, $\mathbf{w}_1^\kappa \in \mathcal{R}^{H\times 4H}$, and $\mathbf{w}_2^\kappa \in \mathcal{R}^{4H\times H}$. During the prefill phase, the input of the κ-th layer is specified by $\mathbf{x}^\kappa$, and `key`, `value`, `query`, and `output` of the attention layer is specified $\mathbf{x}_K^\kappa, \mathbf{x}_V^\kappa, \mathbf{x}_Q^\kappa, \mathbf{x}_{\text{Out}}^\kappa \in \mathcal{R}^{b\times s\times H}$. First, the cached `key`, `value` can be computed by:

$$\mathbf{x}_K^\kappa = \mathbf{x}^\kappa \cdot \mathbf{w}_K^\kappa; \quad \mathbf{x}_V^\kappa = \mathbf{x}^\kappa \cdot \mathbf{w}_V^\kappa$$

The rest of the computation in the κ-th layer is:

$$\mathbf{x}_Q^\kappa = \mathbf{x}^\kappa \cdot \mathbf{w}_Q^\kappa$$
$$\mathbf{x}_{\text{Out}}^\kappa = f_{\text{softmax}}\left(\frac{\mathbf{x}_Q^\kappa {\mathbf{x}_K^\kappa}^T}{\sqrt{H}}\right) \cdot \mathbf{x}_V^\kappa \cdot \mathbf{w}_O^\kappa + \mathbf{x}^\kappa$$
$$\mathbf{x}^{\kappa+1} = f_{\text{relu}}\left(\mathbf{x}_{\text{Out}}^\kappa \cdot \mathbf{w}_1^\kappa\right) \cdot \mathbf{w}_2^\kappa + \mathbf{x}_{\text{Out}}^\kappa$$

During the decode phase, given $\mathbf{t}^\kappa \in \mathcal{R}^{b\times 1\times H}$ as the embedding of the current generated token in the κ-th layer, the inference computation needs to i) update the KV cache:

$$\mathbf{x}_K^\kappa \leftarrow \text{Concat}\left(\mathbf{x}_K^\kappa, \mathbf{t}^\kappa \cdot \mathbf{w}_K^\kappa\right); \mathbf{x}_V^\kappa \leftarrow \text{Concat}\left(\mathbf{x}_V^\kappa, \mathbf{t}^\kappa \cdot \mathbf{w}_V^\kappa\right)$$

And ii) compute the output of the current layer:

$$\mathbf{t}_Q^\kappa = \mathbf{t}^\kappa \cdot \mathbf{w}_Q^\kappa$$
$$\mathbf{t}_{\text{Out}}^\kappa = f_{\text{softmax}}\left(\frac{\mathbf{t}_Q^\kappa \mathbf{x}_K^{\kappa\,T}}{\sqrt{H}}\right) \cdot \mathbf{x}_V^\kappa \cdot \mathbf{w}_O^\kappa + \mathbf{t}^\kappa$$
$$\mathbf{t}^{\kappa+1} = f_{\text{relu}}\left(\mathbf{t}_{\text{Out}}^\kappa \cdot \mathbf{w}_1^\kappa\right) \cdot \mathbf{w}_2^\kappa + \mathbf{t}_{\text{Out}}^\kappa$$

Parallelism in generative inference. Given the scale of the state-of-the-art large language models, distributed or parallel execution is necessary. Two standard parallel strategies are usually included in inference: *Pipeline parallelism* (Huang et al., 2019; Narayanan et al., 2019) partitions the foundation model into multiple stages and serves the inference computation as a pipeline, where each GPU or (group of GPUs) handles a stage. During the inference computation, the GPU(s) serving stage-(j) needs to `send` the activations to the GPU(s) serving stage-(j+1). For inference computation, pipeline parallelism cannot reduce the completion time for a *single* request since only one stage can be active. However, the communication volume included in pipeline parallelism is much less when compared with tensor model parallelism, which is beneficial for slow GPU connections. *Tensor model parallelism* (Narayanan et al., 2021) partitions the inference computation at the level of transformer layers over multiple GPUs, where the weight matrices are distributed both row-wisely and column-wisely. Two `AllReduce` operations are required to aggregate each layer's output activations. Tensor model parallelism splits both the data scan and computation among a tensor model parallel group, which can effectively scale out the inference computation if the connection is fast among the group. However, when the intra-group communication is not extremely fast (i.e., not by NVLink or PCIe), tensor model parallelism can perform poorly.

3. Asymmetric Parallel Implementation

The current foundation model service systems (such as huggingface Accelerate (HuggingFace, 2022), and FastTransformer (NVIDIA, 2022)) can only support a symmetric inference setup — it has to set all the tensor model parallel groups to share the same degree for each pipeline stage serving the same number of transformer layers. We first illustrate a case study about why asymmetric parallel support is necessary under heterogeneous environments and enumerate our system implementation of this functionality.

3.1. Case Study: Parallelism over Heterogeneity

Consider a heterogeneous environment where we can use three GPU instances to serve a LLAMA-2 (70B) model: the first instance has `4×A6000-48G`, the second instance has `2×A5000-24G`, and the third instance has `2×A4000-16G`. We test a generative inference request with an input length of 128, and output length of 64. The results are shown in Figure 1. We have some interesting observations: First, *direct usage of pure tensor model parallel parallelism (TP) or pipeline parallelism (PP) will cause*

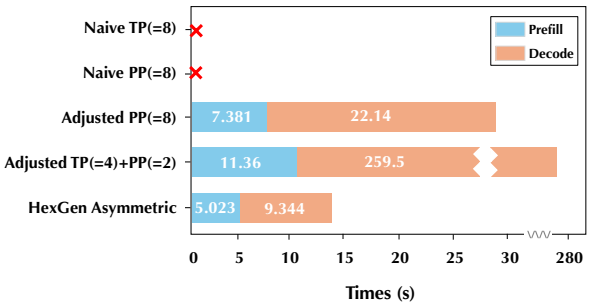

Figure 1: Case study of parallel strategy over heterogeneity.

the out-of-memory (OOM) error: for tensor model parallelism, `A4000-16G` cannot host the evenly partitioned parameter shards; for naive pipeline parallelism, one need to evenly partition the transformer layers, where `A4000-16G` cannot hold 10 layers of LLAMA-2 (70B) model on its stage. Second, *simply partitioning pipeline stage according to computation capacity leads to poor performance*: we test two layouts: i) set PP degree to 8, and partition layer number proportionally with the GPU capacity, in this case, since the pipeline is long where only one stage can be active for computation, leading to slow inference; ii) set PP degree to 2, and TP degree to 4, where we use the first instance `4×A6000-48G` to serve the first stage handling 56 layers and the second and third instances `2×A5000-24G`+`2×A4000-16G` to handle the rest 24 layers, in this case, the tensor model parallelism introduces significant cross-machine communication that compromises the performance. Lastly, *fully asymmetric allocation unleashes the performance under heterogeneity*: to avoid the issue we encounter above, we show the performance of the HEXGEN, where we let each machine serve a single stage and handle 48, 20, 12 layers with TP degrees as 4, 2, 2, respectively — we observe 2× and 19× speedup compared with the symmetric parallel executions.

3.2. HEXGEN Asymmetric Parallel Implementation

In order to implement the asymmetric parallel support, our essential change can be summarized as follows: *each pipeline parallel stage can be assigned with a different number of layers and tensor model parallel degree*. From the perspective of the implementation, we make the following changes: i) when initializing the pipeline parallel communication group, HEXGEN will take the configured tensor model parallel degree for each stage along with their corresponding number of allocated transformer layers; ii) each stage will select a leader GPU which introduces lowest communication latency to GPUs in the nearby stages, and initialize an independent tensor model parallel group; iii) when executing the inference through the pipeline, only the leader GPU in each stage (i.e., tensor model parallel group) will `send` the activation to the leader GPU in the next stage; once the leader GPU `receives` the activation, it will `broadcast` this activation among its tensor model

parallel group to execute the tensor model parallel computation. In terms of system implementation, we modify the latest FlashAttention (Dao, 2023) framework by integrating this new pipeline parallel design to enable this functionality.

4. Scheduling over Heterogeneity

We introduce our scheduling algorithm in this Section.

4.1. Problem Formalization

Notations. We first introduce the following notations: Let $\mathbf{D} = \{d_1 \dots d_N\}$ be a set of N GPU devices, where M_d denotes GPU memory limit, m_d denotes GPU memory bandwidth, and c_d denotes tensor core computation power; $\mathbf{A} \in \mathbb{R}_+^{N\times N}$ and $\mathbf{B} \in \mathbb{R}_+^{N\times N}$ be the communication matrix between these devices describing the latency and bandwidth respectively, where the latency and bandwidth between device d and d' is $\alpha_{d,d'}$ and $\beta_{d,d'}$; L denotes the total number of layers in the model to be served. We summarize all notations used in this paper in Appendix A for easy reference.

Formalization of the scheduling problem. Given the above notations, we can formalize our scheduling problem as follows: suppose $\mathbf{d}_{i,j}$ is a subset of GPU devices that satisfies $\bigcup_{i,j} \mathbf{d}_{i,j} \subseteq \mathbf{D}$, and $\mathbf{d}_{i,j} \bigcap \mathbf{d}_{i',j'} = \emptyset, \quad \forall i \neq i' \vee j \neq j'$, where we suppose that the union of subsets of GPUs $\bigcup_j \mathbf{d}_{i,j}$ serves the i-th model replica as an independent pipeline, and the set of GPUs $\mathbf{d}_{i,j}$ serves the j-th stage in the i-th pipeline that holds $l_{i,j}$ transformer layers, if $\|\mathbf{d}_{i,j}\| > 1$, we indicate that the subset of GPUs $\mathbf{d}_{i,j}$ runs tensor model parallelism. An *assignment* σ is a mapping: $\mathbf{D} \to \{(\mathbf{d}_{i,j}, l_{i,j})\}$, corresponding to a layout of multiple inference worker groups to serve multiple replicas of the same model simultaneously. Consider a set of inference tasks $\mathbf{T}$ that satisfy some distribution[1] $\mathbf{T} \sim \mathcal{P}$. An *optimal assignment* σ^* can be defined as:

$$\begin{aligned} \sigma^* = \arg\max_{\sigma\in\Sigma} \quad & \mathbb{E}_{\mathbf{T}\sim\mathcal{P}} \left[\text{SLO}\left(\text{C}_{\text{comm}}(\sigma) + \text{C}_{\text{comp}}(\sigma)\right)\right] \\ s.t. \quad & \text{C}^d_{\text{mem}}(\sigma) \leq M_d \quad \forall d \in \mathbf{D}. \end{aligned} \tag{1}$$

where $\text{C}_{\text{comm}}(\sigma)$, $\text{C}_{\text{comp}}(\sigma)$, $\text{C}^d_{\text{mem}}(\sigma)$ represents communication cost, computation cost, and memory limit for layout σ. Intuitively, the scheduling problem is to *find an optimal assignment that partitions the device set to represent multiple independent inference pipeline groups that can maximize the inference service level objective (SLO) considering the computation cost, communication cost, and memory consumption constraints*. We summarize the estimation of the computation cost, communication cost, and memory consumption constraints in Table 1, and the detailed formulation is enumerated in Appendix B. As one can see, solving this problem is obviously NP-hard; thus, we adopt a *two-phase* search algorithm to tackle the problem:

[1]The most commonly-used case is Poisson distribution; while one can switch it for any particular distributions.

- We first generate a random partition of $\{\mathbf{d}_{i,\sim}\}$ of $\mathbf{D}$ where $\mathbf{d}_{i,\sim}$ is a set of GPU devices that can be leveraged to serve an independent pipeline group; given this partition, we determine the optimal layout of pipeline stage partition by an efficient dynamic programming method (Section 4.2).
- We propose an evolutionary algorithm that generates the random partition to be used in the first step, where we first define some hard constraints to reduce the search space of the random partition, and then we define some effective mutation operations that can be used to accelerate the convergence of the searching algorithm (Section 4.3).

4.2. Optimizing the Layout of a Pipeline

We first introduce how to solve a sub-optimization problem to find the optimal layout for an independent pipeline. Formally, given a set of GPUs $\mathbf{d}_{i,\sim}$, and a particular layer partition represented by $\{l_{i,j}, j = 1, 2, ..., S_i\}$, this sub-optimization aims to find a local optimal assignment $\mathbf{d}_{i,\sim} \to \{(\mathbf{d}_{i,j}, l_{i,j})\}$ that minimizes the end-to-end inference cost over the i-th pipeline defined in Equation 2 below:

$$\begin{aligned} & \text{C}^i_{\text{comp}}(\{\mathbf{d}_{i,\sim}\}) + \text{C}^i_{\text{comm}}(\{\mathbf{d}_{i,\sim}\}) \\ = & \sum_{j=1}^{S_i} \text{C}^{i,j}_{\text{comp}}(\mathbf{d}_{i,j}) + \sum_{j=1}^{S_i} \text{C}^{i,j}_{\text{comm-tp}}(\mathbf{d}_{i,j}) + \sum_{j=1}^{S_i-1} \text{C}^{i,j}_{\text{comm-pp}}(\mathbf{d}_{i,j}) \\ s.t. \quad & \text{C}^d_{\text{mem}}(\{\mathbf{d}_{i,\sim}\}) \leq M_d \quad \forall d \in \bigcup_j \mathbf{d}_{i,j} \end{aligned} \tag{2}$$

As one can notice, this sub-optimization problem is still NP-hard. To reduce the search space, we adopt the following heuristic: we force each tensor model parallel group to use the *same type* of GPUs on the *same* machine to avoid extensive cross-machine communication overhead. Following this heuristic, we use an additional notation to represent any GPU set by a vector $\boldsymbol{\tau} \in \mathcal{R}^{N_T}$, assuming that N_T is the total number of different GPU types, and τ_k represents the number of the k-th type of GPU in this set.

Next, we introduce the following dynamic programming (DP) algorithm to solve this problem. We define a memorization buffer $\text{DP} \in \mathcal{R}^{S_i \times \#_1 \times \#_2 \dots \times \#_{N_T}}$, where S_i is the total number of stages in this pipeline, $\#_k, k = 1, ..., N_T$ is the total number of k-th type of GPUs in the GPU set $\mathbf{d}_{i,\sim}$, the value $\text{DP}[j; \boldsymbol{\tau}]$ represents the communication and computation cost of assigning the first j stage(s) in the GPU set represented by $\boldsymbol{\tau}$ if the memory limit for the first j stage(s) is satisfied[2]. For example, $\text{DP}[1; [2, 0, ..., 0]]$ represents the communication and computation cost of assigning the first stage in two GPUs of the first type. If we initialize the values in DP to $+\infty$ and set $\text{DP}[0; \boldsymbol{\tau}] = 0$, we can derive the transition formula as:

$$\begin{aligned} \text{DP}[j; \boldsymbol{\tau}] = \min_{\tau_k \cdot \mathbf{e}_k \subset \mathbf{d}_{i,\sim}} & \{\text{DP}[j-1; \boldsymbol{\tau} - \tau_k \cdot \mathbf{e}_k] + \\ & \text{C}^{i,j}_{\text{comp}}(\tau_k \cdot \mathbf{e}_k) + \text{C}^{i,j}_{\text{comm}}(\tau_k \cdot \mathbf{e}_k)\} \end{aligned} \tag{3}$$

[2]$\text{DP}[j; \boldsymbol{\tau}] = +\infty$ if the memory limit is violated for any stage.

Table 1: Modeling the generative inference cost and limit.

Description	Notation	Cost Formulation
Computation cost	$\mathrm{C}_{\text{comp}}^{i,j}\left(\mathbf{d}_{i,j}\right)$	$\max_{d\in\mathbf{d}_{i,j}}\left(\frac{12H^2B_{\text{type}}s_t^{\text{out}}}{\lvert\mathbf{d}_{i,j}\rvert m_d}\right)\cdot l_{i,j}+\max_{d\in\mathbf{d}_{i,j}}\left(\frac{24b_t\left(s_t^{\text{in}}+s_t^{\text{out}}\right)H^2}{\lvert\mathbf{d}_{i,j}\rvert c_d}\right)\cdot l_{i,j}$
TP communication cost	$\mathrm{C}_{\text{comm-tp}}^{i,j}\left(\mathbf{d}_{i,j}\right)$	$\max_{d\in\mathbf{d}_{i,j}}\left(\sum_{d'\in\mathbf{d}_{i,j}-\{d\}}\left(\alpha_{d,d'}+\frac{b_ts_t^{\text{in}}HB_{\text{type}}}{\lvert\mathbf{d}_{i,j}\rvert\beta_{d,d'}}\right)\right)\cdot 4l_{i,j}+\max_{d\in\mathbf{d}_{i,j}}\left(\sum_{d'\in\mathbf{d}_{i,j}-\{d\}}\left(\alpha_{d,d'}+\frac{b_tHB_{\text{type}}}{\lvert\mathbf{d}_{i,j}\rvert\beta_{d,d'}}\right)\right)\cdot 4s_t^{\text{out}}l_{i,j}$
PP communication cost	$\mathrm{C}_{\text{comm-pp}}^{i,j}\left(\mathbf{d}_{i,j}\right)$	$\min_{d\in\mathbf{d}_{i,j},d'\in\mathbf{d}_{i,j+1}}\left(\alpha_{d,d'}+\frac{b_ts_t^{\text{in}}HB_{\text{type}}}{\beta_{d,d'}}\right)+\min_{d\in\mathbf{d}_{i,j},d'\in\mathbf{d}_{i,j+1}}\left(\alpha_{d,d'}+\frac{b_tHB_{\text{type}}}{\beta_{d,d'}}\right)\cdot s_t^{\text{out}}$
Memory limit	$\mathrm{C}_{\text{mem}}^{d}\left(\mathbf{d}_{i,j}\right)$	$\left(\frac{12H^2B_{\text{type}}}{\lvert\mathbf{d}_{i,j}\rvert}+\frac{2b_t\left(s_t^{\text{in}}+s_t^{\text{out}}\right)HB_{\text{type}}}{\lvert\mathbf{d}_{i,j}\rvert}\right)\times l_{i,j}+\quad 4b_t\left(s_t^{\text{in}}+s_t^{\text{out}}\right)HB_{\text{type}}$

We formulate computation cost, tensor model parallel (TP) communication cost, memory limit of the j-th stage in the i-th pipeline, and the pipeline parallel (PP) communication cost between the j-th stage and the $j+1$-th stage of the i-th pipeline for a particular inference task $t\in\mathbf{T}$, where b_t is the batch size, s_t^{in} is the sequence length of input prompt and s_t^{out} is the number of output tokens, and B_{type} denotes the number of bytes for the precision of inference computation, e.g., $B_{\text{type}}\left(\text{FP16}\right)=2$. (Details in Appendix B)

where $\tau_k\cdot\mathbf{e}_k$ is the vector representation of a set of τ_k k-th type GPUs, $\mathrm{C}_{\text{comp}}^{i,j}\left(\tau_k\cdot\mathbf{e}_k\right)$ is defined by Table 1, and $\mathrm{C}_{\text{comm}}^{i,j}\left(\tau_k\cdot\mathbf{e}_k\right)$ is the sum of tensor parallel communication cost and pipeline parallel communication cost follow the formulation in Table 1. Notice these two value will be set to $+\infty$ if the memory limit $\mathrm{C}_{\text{mem}}^{d}\left(\tau_k\cdot\mathbf{e}_k\right)$ defined in Table 1 is violated. Formally, we formalize a recursive implementation of the dynamic programming algorithm in Algorithm 1 to estimate the minimal cost of this pipeline — the assignment of the pipeline stage can be implemented by a standard back-track process over the memorization buffer DP straightforwardly, which we do not enumerate here.

Complexity analysis. The computation of the cost defined in Table 1 can be done in constant time. For each stage, the proposed DP algorithm will visit at most $\prod_{k=1}^{N_T}\#_k$ different subsets of GPUs; the maximal depth of the recursion is S_i; thus, the total time complexity of the proposed DP algorithm is $\mathbb{O}\left(S_i\cdot\prod_{k=1}^{N_T}\#_k\right)$. This is much more efficient than the vanilla approach based on enumeration without the heuristic of forcing tensor parallelism to use the same type of GPUs — in that case, each stage has to consider $2^{\lvert\mathbf{d}_{i,\sim}\rvert}$ different subsets of GPUs, where the total time complexity grows to $\mathbb{O}\left(S_i\cdot 2^{\lvert\mathbf{d}_{i,\sim}\rvert}\right)$. In practice, even a highly heterogeneous GPU pool probably only includes a limited variety of GPU types. The proposed DP algorithm can solve the sub-optimization problem efficiently. This process can be further accelerated by limiting the degree of tensor model parallelism to a smaller candidate set, e.g. $\{1,2,4,8\}$, which reduces the total time complexity to $\mathbb{O}\left(S_i\cdot 4^{N_T}\right)$.

4.3. Searching via Genetic Algorithm

Next, we introduce the genetic algorithm to solve the global optimization problem. Concretely, given the global set of GPUs $\mathbf{D}$, this genetic optimization problem finds an optimal partition of $\mathbf{D}$, to some independent pipeline groups $\mathbf{d}_{i,\sim}$ along with its stage partition $\{l_{i,1},...,l_{i,j},l_{i,S_i}\}$. We enumerate the details of this genetic algorithm as follows.

Initialization. A good initialization of the genetic algorithm is usually helpful in accelerating the whole search process. Thus, we initialize the population with a simple heuristic based on the communication condition. Formally, given the communication matrix $\mathbf{A}$ and $\mathbf{B}$, we execute the vanilla K-means algorithm to construct M independent pipeline parallel groups, where the hyper-parameter M is determined by the standard Elbow method. Intuitively, this helps the assignment avoid using slow cross-region communication links. Note that M is not fixed after initialization, as we can change this by the mutation operations introduced next.

Algorithm 1 Estimate optimal pipeline cost.

```
Input: Memorization Buffer DP,
       Current stage j = 1,
       Assigned GPU set τ = ∅,
       Unassigned GPU set τ̄ = d_{i,∼}
function ESTIMATE-PIPELINE-COST(DP, j, τ, τ̄)
/* N_T is number of different GPU types. */
for k = 1 to N_T do
  /* #_k is number of type k-th GPU in τ̄. */
  for τ_k = 1 to #_k do
    /* The current GPU set δ. */
    δ ← τ_k · e_k;
    /* Compute current cost C. */
    C ← DP[j−1; τ] + C^{i,j}_comp(δ) + C^{i,j}_comm(δ)
    /* Update global memorization buffer. */
    if C < DP[j; τ + δ] then
      DP[j; τ + δ] ← C;
    end if
    /* Recursively assign the next stage(s). */
    if j < S_i and τ̄−δ ≠ ∅ then
      ESTIMATE-PIPELINE-COST(DP, j+1, τ+δ, τ̄−δ)
    end if
  end for
end for
end function
```

Mutation operations. We follow the vector representation of a set of GPUs introduced earlier in Section 4.2 and define three mutation operations in the genetic algorithm.

- `Merge`: Merge two pipeline groups into a single group. Formally, given two independent pipeline groups noted as $\boldsymbol{\tau}^1$ and $\boldsymbol{\tau}^2$, *merge* is defined as: $\boldsymbol{\tau}^1,\boldsymbol{\tau}^2\rightarrow\boldsymbol{\tau}$, where we

have $\boldsymbol{\tau} = \boldsymbol{\tau}^1 + \boldsymbol{\tau}^2$.

- `Split`: Split a single pipeline group to two pipeline groups evenly. Given any independent pipeline group noted as $\boldsymbol{\tau}$, *split* is defined as: $\boldsymbol{\tau} \rightarrow \boldsymbol{\tau}^1, \boldsymbol{\tau}^2$, where for any GPU type indexed by k, we have $\tau_k^1 = \lfloor \frac{\tau_k}{2} \rfloor$, $\tau_k^2 = \lceil \frac{\tau_k}{2} \rceil$.

- `Swap`: Move one GPU from one pipeline group to another pipeline group. Formally, given any two independent pipeline groups noted as $\boldsymbol{\tau}^1$ and $\boldsymbol{\tau}^2$, *swap* is defined as: $\boldsymbol{\tau}^1, \boldsymbol{\tau}^2 \rightarrow \boldsymbol{\tau}'^1, \boldsymbol{\tau}'^2$, where for a particular sampled GPU type index $\widehat{k}$, we have
$$\begin{cases} \tau_k'^1 = \tau_k^1, \quad \tau_k'^2 = \tau_k^2 & \text{if } k \neq \widehat{k}; \\ \tau_k'^1 = \tau_k^1 + 1, \ \tau_k'^2 = \tau_k^2 - 1 & \text{otherwise.} \end{cases}$$

Notice that we also conduct some early checks for the off-springs generated by the above mutation operations. For example, after a `split` operation, if the total device memory of any of these two pipeline groups cannot even hold a copy of model parameters, we can simply remove this off-spring without running the dynamic programming algorithm (Algorithm 1) to accelerate search.

Determine the pipeline partitions. We adopt a simple expectation maximization algorithm to determine any particular pipeline stage partition (i.e. $\{l_{i,1}, ..., l_{i,j}, l_{i,S_i}\}$). Concretely, we first generate an even partition for any new off-spring generated by the mutation operations ($l_{i,j} = \frac{L}{S_i}, j = 1, 2, ..., S_i$). Then after running the dynamic programming algorithm (Algorithm 1), we adjust the pipeline partition proportional to the total memory of the GPU set that currently serves this stage. We find that this heuristic is effective to find a good pipeline partition.

Put it together. We adopt a standard iterative procedure of the genetic algorithm: for each iteration, we conduct the mutation operation(s), run the dynamic algorithm to determine the optimal pipeline assignments, and adjust the pipeline partitions for the current off-springs. To estimate the expected SLO, we adopt the inference task simulator from AlpaServe (Li et al., 2023).

5. Evaluation

To evaluate the design and implementation of HEXGEN, we ask the following essential questions:

- *What is the cost performance trade-off between the centralized homogeneous deployment and* HEXGEN *over a heterogeneous decentralized GPU pool?*

- *What is the end-to-end performance comparison between* HEXGEN *and the state-of-the-art centralized and decentralized generative inference systems?*

- *How effective is the scheduling algorithm in finding the optimal assignment of the inference workflow?*

5.1. Experimental Setup

Runtime. We perform evaluation in the following setups:

- Homogeneous GPUs in a centralized data center. We rent two *AWS* on-demand `p4d.24xlarge` instances, each equipped with `8×NVIDIA A100-40G` GPUs with a budget of $65.54/hour to represent the standard homogeneous case in a data center.

- Heterogeneous GPUs across data centers. We rent GPUs from *FluidStack*, a GPU cloud provider with services for various GPUs. Concretely, we considered two settings based on *real availability*: i) heterogeneous-full-price: we rent two `3090Ti×8` instances in Iceland, two `3090Ti×3` instances in Norway, one `A5000×8` in Nevada, two `A6000×8` instances, one `A5000×8` instances and one `A40×4` instances in Illinois with a budget of $65.04/hour; ii) heterogeneous-half-price: we rent two `3090Ti×8` instances in Iceland, two `3090Ti×3` instance in Norway, and one `A5000×8` instance in Nevada with a budget of $29.6/hour.[3]

Baselines. We carefully select state-of-the-art approaches as baselines. To understand the cost performance trade-offs and system efficiency of HEXGEN, we compare: i) HEXGEN under heterogeneous-full-price, ii) a truncated version of HEXGEN without asymmetric parallel support under heterogeneous-full-price (the allocation of model replicas is still scheduled by our proposed search algorithm), iii) HEXGEN under heterogeneous-half-price, and iv) FLASHATTENTION (Dao, 2023) under homogeneous-data-center setting.[4] To understand end-to-end performance, we compare HEXGEN with HUGGINGFACE-TGI (HuggingFace, 2023) as the state-of-the-art approach under the homogeneous setting and PETALS (Borzunov et al., 2023) as the state-of-the-art approach under decentralized heterogeneous setting. To understand the efficiency of the proposed scheduling algorithm, we compare its convergence with a strawman policy based on random mutation.

Evaluation metrics. Following the generative inference evaluation setup from AlpaServe (Li et al., 2023), we test system performance based on SLO attainment, refers to the proportion of requests that can be finished within a predefined performance threshold. Specifically, we measure SLO attainment in terms of the percentage of requests served within the time frame set by the SLO. We scale the SLO to various multiples of the execution latency of A100

[3] In the cross data center setting, we measure the network latency and bandwidth between GPUs in different regions by configuring a virtual private network through UDP hole punching, and benchmark the NCCL performance: the intra-region latency and bandwidth were 2 ms and 5 Gbps, while inter-region latency and bandwidth range from 40 - 150 ms and 0.3 - 1.0 Gbps.

[4] Our implementation is identical to the standard FLASHATTENTION implementation under a homogeneous setting.

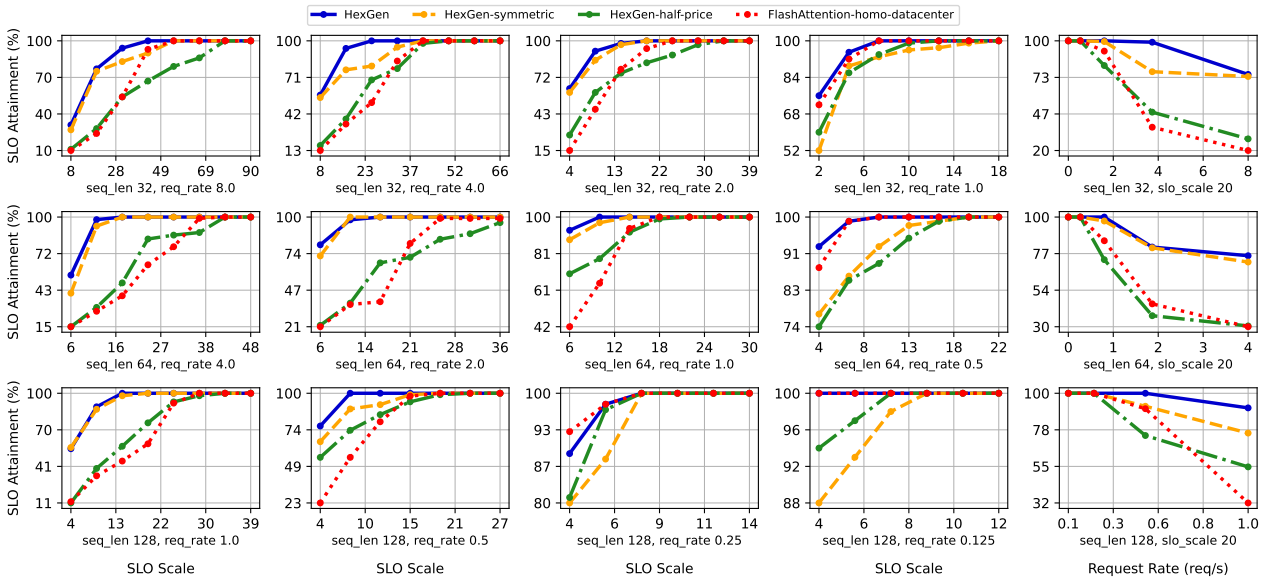

Figure 2: SLO attainment results to evaluate cost performance trade-offs. Each row corresponds to a particular output sequence length (32, 64, 128). The first four columns correspond to different SLO scales ranging from 8 to 0.125 requests per second. The last column represents the performance comparison of various settings at different request rates.

GPUs (SLO Scale in Figure 2), which allows us to evaluate system performance under different levels of operational stringency. We generate inference workload according to a Poisson process parameterized by the request rate. The consecutive requests (inter-arrival times) follow an exponential distribution. For a target SLO goal (e.g., 99%), we focus on two metrics: i) the minimum latency deadline required to achieve the desired attainment, and ii) the system's resilience to peak request rate. We apply the most popular open-source LLAMA-2 (70B) model on some real-world prompts (Lmsys, 2023), and test output sequence lengths from 32 to 128, and request rates varying between 0.125 - 10 requests per second.

5.2. Cost Performance Trade-off

Figure 2 illustrates a comprehensive comparison of the cost performance trade-off in terms of SLO attainment among HEXGEN w/wo asymmetric parallel group support under the full budget in the heterogeneous setting, HEXGEN under the half budget in the heterogeneous setting, and FLASHATTENTION in the homogeneous setting. We want to highlight some interesting results.

Cost efficiency. When given the relatively same budget, HEXGEN under the full budget in the heterogeneous setting clearly outperforms FLASHATTENTION in the homogeneous datacenter setting. In fact, HEXGEN reaches up to 2.3× and on average 1.5× lower latency deadlines, and is capable of handling a peak request rate that is up to 4× higher (2× higher on average). Specifically, by analyzing the scheduling results[5] for heterogeneous-full-price cases, we find that our scheduling approach always prioritizes intra-machine tensor model parallelism to minimize single request latency and employs inter-machine pipeline parallelism to reduce communication over limited bandwidth. It avoids cross-region communication due to ultra-low bandwidth and aims to maximize device memory utilization by incorporating as many model replicas as possible, thereby enhancing parallel request processing. Furthermore, even when we reduce the budget in the heterogeneous setting by half, HEXGEN still reveals similar performance to FLASHATTENTION in the homogeneous setting. We believe that this is strong evidence to illustrate that a decentralized system such as HEXGEN is capable of managing heterogeneous GPUs to provide more economical foundation model inference services without compromising service quality.

Asymmetric parallelism implementation. We also conduct a group of benchmarks to compare HEXGEN w/wo asymmetric parallel group support under the full budget in the heterogeneous setting. The experimental results reveal that the asymmetric parallelism implementation results in up to 1.8× improvement in terms of reaching lower latency deadlines than the original symmetric implementation from FLASHATTENTION. Additionally, the asymmetric parallelism implementation can manage up to 2× higher peak traffic request rate (1.5× on average) compared to the symmetric counterpart. This indicates that besides effective scheduling, the asymmetric parallelism implementation is also necessary to unleash the potential of heterogeneous computational power.

[5]All the strategies chosen by the scheduling algorithm can be found in Appendix F.

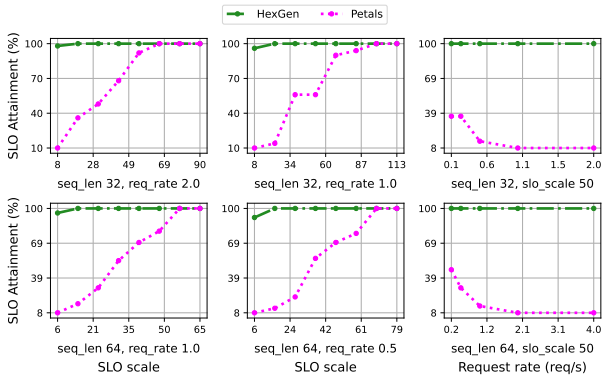

Figure 3: HEXGEN and PETALS. Two rows correspond to output sequence lengths of 32 and 64. First two columns illustrate the results of different SLO scales. The last column shows the effects of request rate on SLO attainment.

5.3. End-to-end System Performance

We compare the end-to-end system performance of HEXGEN with state-of-the-art approaches for heterogeneous (PETALS) and homogeneous (HUGGINGFACE-TGI) settings.

Compare with PETALS. We first compare HEXGEN with PETALS, the state-of-the-art decentralized inference service engine. Figure 3 illustrates the comparison — under the half-price budget in the heterogeneous setting, HEXGEN outperforms PETALS significantly by achieving up to $3.5\times$ lower latency deadline and managing to handle requests at up to $10\times$ higher rates. This is strong evidence to justify the design of HEXGEN: PETALS is mainly built on top of swarm parallelism (Ryabinin et al., 2023), which heavily depends on dynamic adjustment of the collective learning paradigm to ensure the elasticity in a decentralized machine learning system; however, such a dynamic design compromises the inference service performance significantly when comparing with a system like HEXGEN that is equipped with the careful design of static scheduling of the inference workflow.

To evaluate HEXGEN over dynamic GPU pools, we test a scenario where 4 GPUs leave the current allocation scheduled by HEXGEN. In this case, HEXGEN will re-run the search algorithm to find the new optimal allocation. Interestingly, we find this simple policy is very effective—the genetic algorithm is based on local search, which demands much less iteration of searching when only a small portion of GPUs dynamically join or leave. We notice that HEXGEN can rerun the searching algorithm in less than 30 seconds to find the new optimal allocation. Figure 4 illustrates HEXGEN's performance before and after the 4 GPUs become offline. We see the performance gap is considerably small under such dynamics. In addition, we find that the performance of HEXGEN with 4 GPUs offline is still significantly better than PETALS.

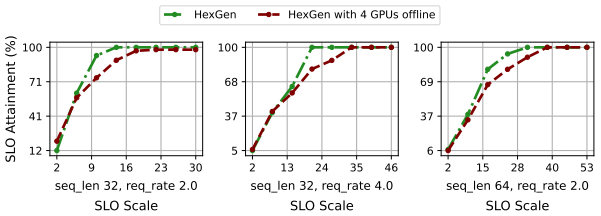

Figure 4: SLO attainment results of HEXGEN compared with HEXGEN with 4 GPUs offline.

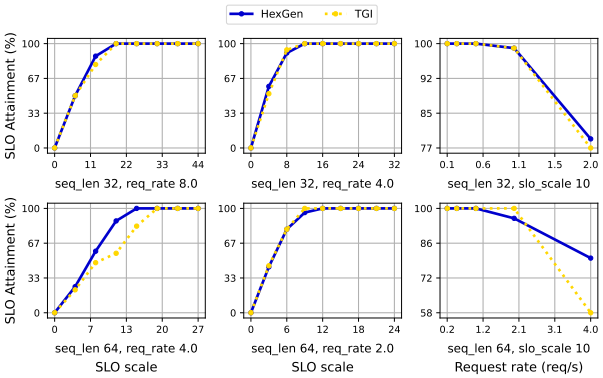

Figure 5: HEXGEN v.s. HUGGINGFACE-TGI. Two rows represent output sequence lengths of 32 and 64. First two columns show the result of different SLO scales. The last column shows the effects of request rate on SLO attainment.

Compare with HUGGINGFACE-TGI. We also compare HEXGEN under the full budget in heterogeneous setting with HUGGINGFACE-TGI in the homogeneous data center settings. In this case, HEXGEN gets almost the same end-to-end performance in both latency and request handling, achieves up to $1.25\times$ lower latency deadlines, and is able to handle requests at the same level of rates.

5.4. Effectiveness of the Scheduling Algorithm

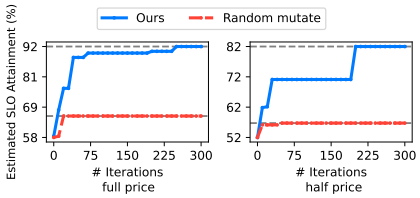

Figure 6: Convergence comparison of the proposed search strategy and random mutation.

To evaluate the effectiveness of the scheduling algorithm, we disable the advanced mutation strategies introduced in Section 4.3 by replacing this part with random mutation and comparing the convergence behavior with the proposed algorithm. We benchmarked the full-price and half-price cases, where we set the output sequence length to 32, and the SLO scale as 5. Figure 6 illustrates the result. The proposed search strategy takes 2.1 and 1.5 minutes to identify

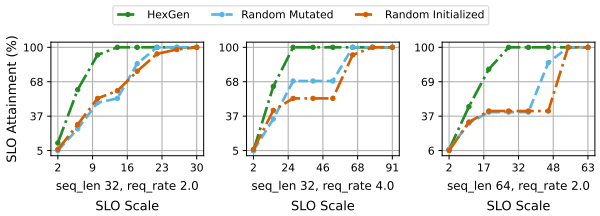

Figure 7: SLO attainment results of HEXGEN compared with random mutation based results and results w/o any searching efforts.

the optimal assignments for full-price and half-price heterogeneous scenarios. This search process runs once before the system is initially deployed, which makes its time cost negligible. We see that our proposed constrained mutation policy significantly outperforms random mutation — the proposed strategy can find assignments that can manage around 26% more SLO attainments than random mutation and converges much faster, while random mutation gets stuck in some local minimum. Additionally, we verified that in both cases, the estimated SLO attainments (92% and 82%) closely align with the actual attainments (94% and 86%) demonstrated in Figure 2. We also provide a direct performance comparison between random initialized allocation (the allocation after executing the initialization method mentioned in subsection 4.3), random mutated policy through evolution and HEXGEN's result in the heterogeneous-half-price scenario. The results are illustrated in Figure 7.

6. Related Work

Foundation model inference optimization. There have been many efforts to accelerate the inference service in terms of both system optimization and algorithm design. On the system side, research efforts have focused on enhancing hardware efficiency through meticulous system optimizations (Fang et al., 2021; Yu et al., 2022; Li et al., 2023; Kwon et al., 2023; Dao et al., 2022). On the algorithm side, some advanced algorithm designs have also been proposed (Leviathan et al., 2023; Yao et al., 2022; Liu et al., 2023), including speculative decoding (Spector & Re, 2023), multiple-head decoding mechanism (Cai et al., 2023) and low precision computation such as quantization (Yao et al., 2022; Frantar et al., 2022; Xiao et al., 2022; Lin et al., 2023), sparsification (Frantar & Alistarh, 2023; Liu et al., 2023) and distillation (Kwon et al., 2022).

Decentralized computation platform. Recently, there have been some emerging research attempts on deploying machine learning computations across a variety of decentralized and heterogeneous computational resources (Ali et al., 2022; Miao et al., 2023b; Zhang et al., 2023; Stoica & Shenker, 2021; Bhat et al., 2023), e.g., distributed training in a collaborative environment (Diskin et al., 2021; Yuan et al., 2022; Ryabinin et al., 2023). However, most of this work does not focus on the system implementation and scheduling for generative inference workflows. Perhaps the most relevant effort is Petals (Borzunov et al., 2022), which allows users to donate different GPUs to perform inference and small-scale fine-tuning. However, Petals is mainly based on dynamic coordination from swarm parallelism (Ryabinin et al., 2023), whose performance is limited by the lack of scheduling of the decentralized inference.

7. Conclusion

In this paper, we explore the opportunity to deploy the inference service of foundation models via a heterogeneous regime with devices of different computation capacities connected over a heterogeneous network. Toward this end, we propose HEXGEN, a generative inference framework with asymmetric parallel support and an effective scheduling algorithm to accommodate such deployment. Our empirical study suggests that when given the same budget, HEXGEN can outperform the centralized homogeneous deployment by either achieving $2.3\times$ lower latency deadlines or tolerating up to $4\times$ more traffic request rate; additionally, HEXGEN also significantly outperforms Petals, the state-of-the-art decentralized collaborative inference prototype by $10\times$ more traffic request rate. We will make the HEXGEN system fully open-sourced and hope that such a system can contribute to the democratization of the usage of foundation models.

Acknowledgments

This work is supported by the HKUST startup grant R9895 from CSE, the UCloud research grant, and the HKUST-WeBank Joint Lab project.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none of which we feel must be specifically highlighted here.

References

Ali, A., Pinciroli, R., Yan, F., and Smirni, E. Optimizing inference serving on serverless platforms. *Proceedings of the VLDB Endowment*, 15(10):2071–2084, 2022.

Athlur, S., Saran, N., Sivathanu, M., Ramjee, R., and Kwatra, N. Varuna: scalable, low-cost training of massive deep learning models. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pp. 472–487, 2022.

Bhat, S., Chen, C., Cheng, Z., Fang, Z., Hebbar, A., Kannan, S., Rana, R., Sheng, P., Tyagi, H., Viswanath, P.,

et al. Sakshi: Decentralized ai platforms. *arXiv preprint arXiv:2307.16562*, 2023.

Bommasani, R., Hudson, D. A., Adeli, E., Altman, R., Arora, S., von Arx, S., Bernstein, M. S., Bohg, J., Bosselut, A., Brunskill, E., et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Borzunov, A., Baranchuk, D., Dettmers, T., Ryabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative inference and fine-tuning of large models. *arXiv preprint arXiv:2209.01188*, 2022. URL https://arxiv.org/abs/2209.01188.

Borzunov, A., Baranchuk, D., Dettmers, T., Riabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative inference and fine-tuning of large models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 558–568, 2023.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

Cai, T., Li, Y., Geng, Z., Peng, H., and Dao, T. Medusa: Simple framework for accelerating llm generation with multiple decoding heads. https://github.com/FasterDecoding/Medusa, 2023.

Cheatham, T., Fahmy, A., Stefanescu, D., and Valiant, L. Bulk synchronous parallel computing—a paradigm for transportable software. *Tools and Environments for Parallel and Distributed Systems*, pp. 61–76, 1996.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Diskin, M., Bukhtiyarov, A., Ryabinin, M., Saulnier, L., Sinitsin, A., Popov, D., Pyrkin, D. V., Kashirin, M., Borzunov, A., Villanova del Moral, A., et al. Distributed deep learning in open collaborations. *Advances in Neural Information Processing Systems*, 34:7879–7897, 2021.

Fang, J., Yu, Y., Zhao, C., and Zhou, J. Turbotransformers: an efficient gpu serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402, 2021.

Frantar, E. and Alistarh, D. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Guo, R., Guo, V., Kim, A., Hildred, J., and Daudjee, K. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. *Proceedings of Machine Learning and Systems*, 4:779–794, 2022.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

HuggingFace. Hugging face accelerate. https://huggingface.co/docs/accelerate/index, 2022.

HuggingFace. Text generation inference. https://huggingface.co/docs/text-generation-inference/index, 2023.

Institute, T. I. Falcon 180b, 2023. URL https://falconllm.tii.ae/falcon-180b.html.

Kwon, S. J., Kim, J., Bae, J., Yoo, K. M., Kim, J.-H., Park, B., Kim, B., Ha, J.-W., Sung, N., and Lee, D. Alphatuning: Quantization-aware parameter-efficient adaptation of large-scale pre-trained language models. *arXiv preprint arXiv:2210.03858*, 2022.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Li, Z., Zheng, L., Zhong, Y., Liu, V., Sheng, Y., Jin, X., Huang, Y., Chen, Z., Zhang, H., Gonzalez, J. E., et al. {AlpaServe}: Statistical multiplexing with model parallelism for deep learning serving. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pp. 663–679, 2023.

LibP2P. A modular network stack, 2023. URL https://libp2p.io/.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B., Song, Z., Shrivastava, A., Zhang, C., Tian, Y., Re, C., et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pp. 22137–22176. PMLR, 2023.

Lmsys. Chatbot arena conversations. https://huggingface.co/datasets/lmsys/chatbot_arena_conversations, 2023.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Wong, R. Y. Y., Chen, Z., Arfeen, D., Abhyankar, R., and Jia, Z. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023a.

Miao, X., Shi, Y., Yang, Z., Cui, B., and Jia, Z. Sdpipe: A semi-decentralized framework for heterogeneity-aware pipeline-parallel training. *Proceedings of the VLDB Endowment*, 16(9):2354–2363, 2023b.

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 1–15, 2019.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer, 2022.

Ryabinin, M., Dettmers, T., Diskin, M., and Borzunov, A. Swarm parallelism: Training large models can be surprisingly communication-efficient. *arXiv preprint arXiv:2301.11913*, 2023.

Spector, B. F. and Re, C. Accelerating llm inference with staged speculative decoding. In *Workshop on Efficient Systems for Foundation Models@ ICML2023*, 2023.

Stoica, I. and Shenker, S. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 26–32, 2021.

Thorpe, J., Zhao, P., Eyolfson, J., Qiao, Y., Jia, Z., Zhang, M., Netravali, R., and Xu, G. H. Bamboo: Making preemptible instances resilient for affordable training of large {DNNs}. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 497–513, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Xiao, G., Lin, J., Seznec, M., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.

Yang, Z., Wu, Z., Luo, M., Chiang, W.-L., Bhardwaj, R., Kwon, W., Zhuang, S., Luan, F. S., Mittal, G., Shenker, S., et al. {SkyPilot}: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pp. 437–455, 2023.

Yao, X. Open Compute Framework: Peer-to-Peer Task Queue for Foundation Model Inference Serving, September 2023. URL https://github.com/autoai-org/OpenComputeFramework.

Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*, 2022.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. Orca: A distributed serving system for {Transformer-Based} generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Yuan, B., He, Y., Davis, J., Zhang, T., Dao, T., Chen, B., Liang, P. S., Re, C., and Zhang, C. Decentralized training of foundation models in heterogeneous environments. *Advances in Neural Information Processing Systems*, 35: 25464–25477, 2022.

Zhang, Q., Li, J., Zhao, H., Xu, Q., Lu, W., Xiao, J., Han, F., Yang, C., and Du, X. Efficient distributed transaction processing in heterogeneous networks. *Proceedings of the VLDB Endowment*, 16(6):1372–1385, 2023.

Contents: In Section A, we summarize the notations throughout this paper for easy reference. In Section B, We enumerate how we formulate the generative inference cost in terms of computation time and communication time in the tensor model and pipeline parallelism and the corresponding memory limit. In Section C, we present an extended discussion on HEXGEN system implementation about the task coordinator. In Section D, we discuss the limitation of batching implementation for HEXGEN. In Section E, we elaborate on the extended version of related works.

A. Summarization of Notations

We summarize the notations used for the scheduling algorithm of this paper in Table 2 for easy reference.

Table 2: Table of notations.

Symbol	Description
N	Total number of GPUs to serve generative inference.
N_T	Total number of distinct GPU types.
$\mathbf{D}$	Set of N GPU devices.
M_d	Memory limit of device d.
m_d	GPU memory bandwidth of device d.
c_d	Tensor core computation power of device d.
$\mathbf{A}$	Communication matrix between devices describing the latency.
$\mathbf{B}$	Communication matrix between devices describing the bandwidth.
$\alpha_{d,d'}$	Latency between devices d and d'.
$\beta_{d,d'}$	Bandwidth between devices d and d'.
L	Total number of layers in the model to be served.
H	Size of the hidden dimension in a transformer block.
B_{type}	Byte size for computational precision.
s_t^{in}	Length of input sequence for task t.
s_t^{out}	Length of output sequence for task t.
b_t	Batch size allocated for inference task t.
$\mathbf{d}_{i,j}$	Set of GPUs serves the j-th stage in the i-th pipeline.
$l_{i,j}$	Number of transformer layers in the j-th stage of the i-th pipeline.
σ	Mapping of models to devices.
$\mathbf{T}$	Set of inference tasks.
$\mathcal{P}$	Distribution of inference tasks.
S_i	Number of stages in the i-th pipeline.
k	Identifier for a GPU type.
$\#_k$	Total number of k-th type of GPUs in the GPU set $\mathbf{d}_{i,\sim}$.
$\boldsymbol{\tau}$	Vector denoting any GPU set.
τ_k	Number of the k-th type of GPU in $\boldsymbol{\tau}$.
$\tau_k \cdot \mathbf{e}_k$	Vector denoting a set of τ_k k-th type GPUs, where $\mathbf{e}_k$ denotes k-th standard basis vector.
M	Number of independent pipeline parallel groups.

B. Modeling the Generative Inference Cost

We enumerate how we formulate the generative inference cost in terms of computation time and communication time in the tensor model and pipeline parallelism and the corresponding memory limit. Following the notation introduced in Sections 2 and 4.1. Let B_{type} be the number of bytes for the precision of inference computation, e.g., $B_{\text{type}}\,(\text{FP}16) = 2$; for a particular inference task $t \in \mathbf{T}$, where b_t is the batch size, s_t^{in} be the sequence length of input prompt and s_t^{out} be the sequence length of output token generation. Comprehensively, given a particular assignment σ output noted as $\{\mathbf{d}_{i,j}\}$, we estimate the communication cost, the computation cost, and the memory limit as follows:

Model the computation time. Recall the introduction of inference computation in Section 2; one can notice that most of the computation is spent on matrix multiplications in a transformer block, while only a very small portion of computation is spent on other components, such as no-linear activation functions, batch normalization, etc. To simplify the modeling, we model the computation time that mainly comes from two sources: (i) *scanning the model parameters through GPU high memory bandwidths to tensor cores*, since usually $b_t \ll H$, we ignore the time to scan the intermediate computation results; (ii) *the computation w.r.t the matrix multiplications in the transformer block*, here we assume the computation time is only determined by the total float number operations in the transformer block and the devices' peak FLOPS, ignores other potential dynamic factors that can influence the execution time. Given a particular layer running tensor model parallelism

over a set of GPU devices noted as $\mathbf{d}_{i,j}$ (j-th stage in i-th pipeline), the computation time of $l_{i,j}$ layer of transformers noted as $\mathbf{C}_{\text{comp}}^{i,j}\left(\mathbf{d}_{i,j}\right)$ can be determined by the following formula:

$$\mathbf{C}_{\text{comp}}^{i,j}\left(\mathbf{d}_{i,j}\right)=\max_{d\in\mathbf{d}_{i,j}}\left(\frac{12H^2B_{\text{type}}s_t^{\text{out}}}{|\mathbf{d}_{i,j}|\,m_d}\right)\cdot l_{i,j}+\max_{d\in\mathbf{d}_{i,j}}\left(\frac{24b_t\left(s_t^{\text{in}}+s_t^{\text{out}}\right)H^2}{|\mathbf{d}_{i,j}|\,c_d}\right)\cdot l_{i,j} \tag{4}$$

where the first part is to estimate the memory scan cost — the model parameter has to be scanned s_t^{out} times for all the generated tokens; $12H^2$ represents the total number of parameters in a transformer layer (e..g, the κ-th layer includes the weights $\mathbf{w}_K^\kappa,\mathbf{w}_Q^\kappa,\mathbf{w}_V^\kappa,\mathbf{w}_O^\kappa\in\mathcal{R}^{H\times H}$, $\mathbf{w}_1^\kappa\in\mathcal{R}^{H\times 4H}$, and $\mathbf{w}_2^\kappa\in\mathcal{R}^{4H\times H}$); the second part is about the matrix multiplication — for each layer, there are $\frac{24b_ts_t^{\text{in}}H^2}{|\mathbf{d}_{i,j}|c_d}$ float point operations for the prefill phase and $\frac{24b_tH^2}{|\mathbf{d}_{i,j}|c_d}$ float point operations for each token. To be more specific, the total number of float point operations for a matrix multiplication between matrix $\mathbf{X}\in\mathcal{R}^{H_1\times H_2}$ and matrix $\mathbf{Y}\in\mathcal{R}^{H_2\times H_3}$ can be calculated by $2H_1H_2H_3$ — $\frac{24b_ts_t^{\text{in}}H^2}{|\mathbf{d}_{i,j}|c_d}$ and $\frac{24b_tH^2}{|\mathbf{d}_{i,j}|c_d}$ are calculated by accumulating the float point operations of matrix multiplications in the prefill phase and for one generated token in the decoding phrase as we introduced in Section 2. Notice that if $\|\mathbf{d}_{i,j}\|=1$, Equation 4 still holds, which represents only one GPU serves this stage without running tensor model parallelism.

Model the communication time. We make the following assumptions about modeling the communication cost: (i) for the point-to-point communication, we use the $\alpha-\beta$ cost model, where the communication time can be estimated by $\alpha+\frac{B}{\beta}$, and B is the number of bytes that needs to be communicated; (ii) for collective communications, we adopt the bulk synchronous parallel (BSP) model (Cheatham et al., 1996) — the communication execution is subdivided into supersteps, each associated with a global synchronization; and the total cost of a superstep is the *max* over all processors at that superstep.

We first model the communication in **tensor model parallelism**. In tensor model parallelism, each GPU needs to conduct two `AllReduce` operations for each transformer block. By the BSP model, we assume the tensor that needs to be aggregated by `AllReduce` will be partitioned to equal chunks, and accomplished by two supersteps (phases) (`ReduceScatter` and `AllGather`), in the `ReduceScatter` phase, each GPU sends its chunk to every other GPU and conduct the aggregation for its chunk; in the `AllGather` phase, each GPU sends its aggregated chunk to every other GPU. Formally, given a particular layer running tensor model parallelism over a set of GPU devices noted as $\mathbf{d}_{i,j}$ (j-th stage in i-th pipeline), the tensor model parallel communication time of l_i,j layer of transformers noted as $\mathbf{C}_{\text{comm-tp}}^{i,j}\left(\mathbf{d}_{i,j}\right)$ can be determined by the following formula:

$$\begin{aligned}\mathbf{C}_{\text{comm-tp}}^{i,j}\left(\mathbf{d}_{i,j}\right)=&\max_{d\in\mathbf{d}_{i,j}}\left(\sum_{d'\in\mathbf{d}_{i,j}-\{d\}}\left(\alpha_{d,d'}+\frac{b_ts_t^{\text{in}}HB_{\text{type}}}{|\mathbf{d}_{i,j}|\,\beta_{d,d'}}\right)\right)\cdot 4l_{i,j}\\&+\max_{d\in\mathbf{d}_{i,j}}\left(\sum_{d'\in\mathbf{d}_{i,j}-\{d\}}\left(\alpha_{d,d'}+\frac{b_tHB_{\text{type}}}{|\mathbf{d}_{i,j}|\,\beta_{d,d'}}\right)\right)\cdot 4s_t^{\text{out}}l_{i,j}\end{aligned} \tag{5}$$

Where the first part is to estimate the tensor model parallel communication cost in prefill phase, while the second term corresponds to the decoding phrase. Notice that one can verify that if $\|\mathbf{d}_{i,j}\|=1$, $\mathbf{C}_{\text{comm-tp}}^{i,j}\left(\mathbf{d}_{i,j}\right)=0$, which illustrate that there is no communication cost if only one GPU is serving this stage.

Next, we model the communication cost in **pipeline parallelism**. To estimate the communication cost between nearby stages ($\mathbf{d}_{i,j}$ and $\mathbf{d}_{i,j+1}$) in pipeline parallelism, we model this communication process by using the fastest link between these two stages. Formally, the pipeline parallel communication cost noted as $\mathbf{C}_{\text{comm-pp}}^{i,j}\left(\mathbf{d}_{i,j}\right)$ can be formalized as:

$$\mathbf{C}_{\text{comm-pp}}^{i,j}\left(\mathbf{d}_{i,j}\right)=\min_{d\in\mathbf{d}_{i,j},d'\in\mathbf{d}_{i,j+1}}\left(\alpha_{d,d'}+\frac{b_ts_t^{\text{in}}HB_{\text{type}}}{\beta_{d,d'}}\right)+\min_{d\in\mathbf{d}_{i,j},d'\in\mathbf{d}_{i,j+1}}\left(\alpha_{d,d'}+\frac{b_tHB_{\text{type}}}{\beta_{d,d'}}\right)\cdot s_t^{\text{out}} \tag{6}$$

Where the first part is to estimate the pipeline model parallel communication cost in prefill phase, while the second term corresponds to the decoding phrase.

Model the memory constraint. The GPU memory footprint mainly comes from three sources during inference computation: (i) to store the model parameters; (ii) to store the intermediate results including `key` and `value` for *each* transformer blocks; and (iii) some activation caches—in an efficient implementation, such memory buffer will be reused during the computation for all transformer blocks, in our implementation the number of such buffer is 4. Formally the memory constraint $\mathbf{C}_{\text{mem}}^{d}\left(\mathbf{d}_{i,j}\right)$

for a device $d \in \mathbf{d}_{i,j}$ can be formalized as:

$$\mathbf{C}_{\text{mem}}^{d}\left(\mathbf{d}_{i,j}\right) = \left(\frac{12H^2 B_{\text{type}}}{|\mathbf{d}_{i,j}|} + \frac{2b_t\left(s_t^{\text{in}} + s_t^{\text{out}}\right) HB_{\text{type}}}{|\mathbf{d}_{i,j}|}\right) \times l_{i,j} + 4b_t\left(s_t^{\text{in}} + s_t^{\text{out}}\right) HB_{\text{type}} \tag{7}$$

Performance alignment. To evaluate the accuracy of our cost model, we conduct a suite of micro-benchmarks to validate the output of our cost estimation by comparing it with the actual execution time. The results are listed in Table 3 and suggest that our cost model can align with the actual execution accurately.

Table 3: Comparison of Benchmarked and Estimated Performance Metrics.

Input/Output Length	Parallel Configuration	Prefill Time - Benchmarked	Prefill Time - Estimated	Decode Time - Benchmarked	Decode Time - Estimated
256/32	TP=8	2.72s	2.99s	2.43s	2.46s
	TP=4 PP=2	3.79s	3.85s	2.25s	2.14s
	TP=2 PP=4	5.26s	5.25s	3.29s	3.04s
	PP=8	8.04s	7.83s	6.04s	5.60s
512/64	TP=8	3.04s	3.10s	4.76s	4.92s
	TP=4 PP=2	4.16s	4.10s	4.32s	4.28s
	TP=2 PP=4	5.57s	5.63s	6.65s	6.08s
	PP=8	8.27s	8.49s	12.4s	11.2s

C. Task Coordinator

To deploy HEXGEN in a real heterogeneous decentralized environment, we implement a **task coordinator**. The task coordinator manages the GPUs from the heterogeneous pool and organizes the independent pipeline parallel worker groups according to the optimal allocation based on the approach we introduced in Section 4. Concretely, the task coordinator is mainly based on an open-source implementation of decentralized computation coordination (Yao, 2023) that utilizes libP2P (LibP2P, 2023) to establish connections among the work groups in a peer-to-peer network. When an inference request is received by the task coordinator, the request will be directed to an appropriate worker group according to the scheduling result.

D. Limitation of Batching Implementation

When we compare HEXGEN with HUGGINGFACE-TGI, we realize that one particular reason that actually limits HEXGEN from reaching its full potential when compared with HUGGINGFACE-TGI — state-of-the-art foundation model inference services usually include some advanced batching policy to improve the hardware efficiency significantly; the current version of HEXGEN has not integrated this feature yet since the batching policy under the heterogeneous setting issues some unique challenges given the diversified executing time and memory limit of each independent pipeline groups, which makes the current batching policy difficult to integrate. We acknowledge this limitation in the current version of HEXGEN and leave this as an important future work to further improve HEXGEN's end-to-end inference performance under the heterogeneous setting.

E. Extended Related Works

Foundation model inference optimization. There have been many efforts to accelerate the inference service in terms of both system optimization and algorithm design. On the system side, research efforts have focused on enhancing hardware efficiency through meticulous system optimizations (Fang et al., 2021; Yu et al., 2022; Li et al., 2023; Kwon et al., 2023; Dao et al., 2022). For example, AlpaServe (Li et al., 2023) proposes a concrete analysis of model parallel strategies and model placement to improve inference service efficiency; PagedAttention (Kwon et al., 2023) introduces an advanced memory management system to batch inference jobs inspired by the classic design of virtual memory and paging; FLashAttention (Dao, 2023) leveraged GPU memory hierarchy to reduce GPU memory access significantly, leading to runtime speedup. On the algorithm side, some advanced algorithm designs have also been proposed (Leviathan et al., 2023; Yao et al., 2022; Liu et al., 2023). For example, speculative decoding (Leviathan et al., 2023; Miao et al., 2023a; Spector & Re, 2023) based algorithms improve the system efficiency by leveraging a small approximation model for prediction and the

original model for parallel verification. A similar idea has been explored by Medusa (Cai et al., 2023), which implements a multiple-head decoding mechanism for parallel token verification. Low precision computation has also been explored to speed up generative inference, such as quantization (Yao et al., 2022; Frantar et al., 2022; Xiao et al., 2022; Lin et al., 2023), sparsification (Frantar & Alistarh, 2023; Liu et al., 2023) and distillation (Kwon et al., 2022).

Decentralized computation platform. Recently, there have been some emerging research attempts on deploying machine learning computations across a variety of decentralized and heterogeneous computational resources (Ali et al., 2022; Miao et al., 2023b; Zhang et al., 2023). For example, sky computing (Stoica & Shenker, 2021; Yang et al., 2023) builds an additional layer above classic cloud platforms to access more economic computation resources and enable interoperability between multi-clouds; SAKSHI (Bhat et al., 2023) has proposed a blueprint to advocate for the hosting and delivery of reliable AI models in the landscape of AI services. There have also been some attempts to deploy machine learning training in a collaborative environment (Diskin et al., 2021; Yuan et al., 2022; Ryabinin et al., 2023). However, most of this work does not focus on the system implementation and scheduling for generative inference workflows. Perhaps the most relevant effort is Petals (Borzunov et al., 2022), which allows users to donate different GPUs to perform inference and small-scale fine-tuning. However, Petals is mainly based on dynamic coordination from swarm parallelism (Ryabinin et al., 2023), whose performance is limited by the lack of scheduling of the decentralized inference as we illustrated in Section 5.3.

F. Case Study of the Scheduling Result

We list the partition results generated by HEXGEN in the heterogeneous-full-price scenario. We use the following representation to describe the scheduled results. We use an array to specify one independent inference pipeline, and the number represents the degree of tensor parallelism for this pipeline stage. For example, [4,2] indicates a two-stage pipeline, where the first stage has a tensor parallel degree of 4, and the second stage has a tensor parallel degree of 2. The scheduled results are listed in Table 4.

Partition breakdown. In Iceland, two 8 $\times$ 3090Ti instances deploy [4,4] strategy to support two model replicas, each function within a single machine. Two 3 $\times$ 3090Ti instances in Norway employ a [2,1,1,2] strategy; cross-machine pipeline parallelism communication happens between the 2nd and 3rd stage. In Nevada, 8 $\times$ A5000 GPUs deploy a [4,4] strategy. In Illinois, 12 $\times$ A6000 GPUs serve four model replicas, each with strategy [2,1]. The remaining 4 $\times$ A6000 / 4 $\times$ A40 is split into four groups, each 2 $\times$ A6000 / 2 $\times$ A40 and another 2 $\times$ A5000 deploys a [2,2] strategy, inter-machine communication is finished by pipeline parallelism.

Table 4: GPU Deployment and Strategy by Region.

Region	GPU Configuration	Strategy
Iceland	$8 \times$ 3090Ti	$[4, 4]$
	$8 \times$ 3090Ti	$[4, 4]$
Norway	$3 \times$ 3090Ti $+ 3 \times$ 3090Ti	$[2, 1, 1, 2]$
Nevada	$8 \times$ A5000	$[4, 4]$
Illinois	$3 \times$ A6000	$[2, 1]$
	$3 \times$ A6000	$[2, 1]$
	$3 \times$ A6000	$[2, 1]$
	$3 \times$ A6000	$[2, 1]$
	$2 \times$ A6000 $+ 2 \times$ A5000	$[2, 2]$
	$2 \times$ A6000 $+ 2 \times$ A5000	$[2, 2]$
	$2 \times$ A40 $+ 2 \times$ A5000	$[2, 2]$
	$2 \times$ A40 $+ 2 \times$ A5000	$[2, 2]$

Interesting insight. In a homogeneous setting, the 16 A100 GPUs can serve 4 LLAMA-2 (70B) model replicas. While in a heterogeneous setting, the 58 cloud GPUs with various types can serve a maximum of 12 LLAMA-2 (70B) model replicas with various hybrid parallel configurations within the same budget. Here are some interesting insights:

- In this scenario, although individual inference tasks in a heterogeneous environment may experience increased latency, the overall performance of the system significantly improves.

- Comprehensively, our scheduling approach prioritizes intra-machine tensor model parallelism to minimize single request latency and employs inter-machine pipeline parallelism to reduce communication over limited bandwidth. It avoids cross-region communication due to ultra-low bandwidth and aims to maximize device memory utilization by incorporating as many model replicas as possible.
- Additionally, asymmetric parallelism plays a significant role in enhancing system performance, primarily by allowing the adoption of more adaptable parallel strategies to minimize extensive communication via low-bandwidth links. For instance, consider a scenario with 4 A5000 GPUs on one machine and 2 on another. Given the exceptionally low intercommunication bandwidth and the requirement for at least 6×24 GB GPUs to support a 70B model replica, the optimal configuration involves establishing a tensor model parallel group of 4 as the first pipeline stage and a group of 2 as the second. This setup is preferred because pipeline parallel communication demands are substantially lower than those of tensor model parallelism.