GUIDE: A Global Unified Inference Engine for Deploying Large Language Models in Heterogeneous Environments

Yanyu Chen[1#], Ganhong Huang[2#]
[#]*These authors contributed equally to this work,*
[1]*School of Systems Science and Engineering, Sun Yat-sen University,*
[2]*School of Computer Science and Engineering, Sun Yat-sen University*

Abstract

Efficiently deploying large language models (LLMs) in real-world scenarios remains a critical challenge, primarily due to hardware heterogeneity, inference framework limitations, and workload complexities. These challenges often lead to inefficiencies in memory utilization, latency, and throughput, hindering the effective deployment of LLMs, especially for non-experts. Through extensive experiments, we identify key performance bottlenecks, including sudden drops in memory utilization, latency fluctuations with varying batch sizes, and inefficiencies in multi-GPU configurations. These insights reveal a vast optimization space, shaped by the intricate interplay of hardware, frameworks, and workload parameters. This underscores the need for a systematic approach to optimize LLM inference, motivating the design of our framework, GUIDE. GUIDE leverages dynamic modeling and simulation-based optimization to address these issues, achieving prediction errors between 25% and 55% for key metrics such as batch latency, TTFT, and decode throughput. By effectively bridging the gap between theoretical performance and practical deployment, our framework empowers practitioners—particularly non-specialists—to make data-driven decisions and unlock the full potential of LLMs in heterogeneous environments cheaply.

核心是做 Deployment tuning 的

1 Introduction

The deployment of Large Language Models (LLMs) has become a pressing challenge, driven by their transformative breakthroughs in natural language processing, computer vision, and multimodal tasks. Models such as GPT [1], OPT [2], LLaMA [3], and Qwen [4], equipped with billions or even trillions of parameters, demonstrate unparalleled capabilities in generating semantically coherent, contextually rich content and solving complex tasks. These advancements have enabled widespread adoption across diverse applications, including chatbots, content creation, and scientific research.

However, efficiently deploying such LLMs in real-world scenarios remains a critical bottleneck due to the complexity of real-world constraints. For many enterprise users and individual practitioners, the deployment process is particularly challenging because it requires deep expertise in hardware configurations, inference frameworks, and optimization strategies. Without this expertise, they often struggle to fully leverage the capabilities of LLMs, leading to underutilized hardware, inefficient resource allocation, and suboptimal performance. This complexity underscores the urgent need for accessible and systematic tools that simplify the deployment process, enabling users with varying levels of expertise to achieve efficient and cost-effective inference.

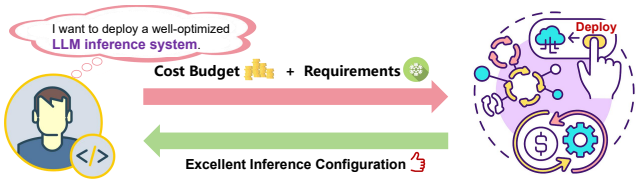

Figure 1: Workflow of the GUIDE system, which helps users input cost and requirements to generate an optimized LLM inference configuration.

Inference performance is heavily influenced by the heterogeneity of hardware platforms, inference frameworks, and deployment strategies. These disparities often result in underutilized computational resources, increased operational costs, and degraded user experiences, such as higher latency or reduced throughput. These challenges underscore the urgent need for systematic optimization in LLM inference, including hardware-aware tuning, inference framework enhancements, and intelligent scheduling, to bridge the gap between theoretical state-of-the-art model performance and practical deployment requirements, while improving resource utilization, reducing latency, and lowering deployment costs.

Despite extensive efforts to optimize LLM inference, existing approaches often target isolated aspects, such as hardware acceleration [5], framework-specific tuning, or parallelization strategies [6].While these methods achieve localized improvements, they fall short in addressing the intricate dependencies

and dynamic interactions among model architectures, hardware platforms, inference frameworks, and optimization techniques. For instance, tensor parallelism improves throughput by distributing computations across GPUs, but the accompanying communication overheads often diminish its effectiveness, particularly for long sequence lengths or small batch sizes [6]. Similarly, inference frameworks like vLLM [7] and Fastgen [8] excel in specific scenarios but face difficulties in adapting to variable workloads or heterogeneous hardware configurations, resulting in suboptimal performance under diverse conditions. These fragmented approaches fail to fully exploit the optimization potential, particularly in heterogeneous deployment environments where performance bottlenecks stem from multi-dimensional factors, including batch size, sequence length, memory capacity, and computational throughput.

To better understand these challenges, we conducted extensive experiments across diverse hardware platforms, inference frameworks, deployment configurations, and optimization methodologies. Our findings reveal significant opportunities for optimization, particularly in memory efficiency and latency reduction, but also underscore the inherent complexity arising from the interplay between hardware, software, workload characteristics, and optimization techniques. For instance, we observed abrupt memory utilization drop-offs under specific configurations, significant latency divergence with varying batch sizes, and performance degradation even in modest parallel scenarios using multiple GPUs. These results highlight the critical need for careful selection and integration of hardware platforms, inference frameworks, deployment configurations, and optimization strategies, tailored to the specific requirements of the workload, to maximize inference efficiency. They also point to limitations in existing deployment practices, which often fail to capture the nuanced interactions among these dimensions, resulting in inefficient resource utilization and suboptimal performance. Addressing these limitations will require a more holistic approach that dynamically balances hardware, software, and optimization techniques to adapt to varying workload demands.

While these insights reveal a vast and complex optimization space, they also highlight the inherent limitations of existing optimization practices. Manual tuning of deployment configurations is resource-intensive, prone to human error, and often lacks the flexibility to generalize across diverse scenarios. Furthermore, while existing data provide valuable insights, they are limited in scope, covering only a subset of frameworks, strategies, and optimization techniques. This narrow coverage hinders comprehensive evaluation of their effectiveness and complicates performance prediction in unseen configurations. This incomplete coverage not only constrains systematic exploration of the optimization space but also inhibits practitioners from effectively adapting to the dynamic evolution of models, frameworks, and hardware.

In response to these challenges, we propose GUIDE, a comprehensive modeling and simulation framework to systematically explore and optimize the inference process of LLMs. By constructing a performance model that incorporates key factors such as hardware (e.g., GPUs), inference frameworks, deployment strategies, optimization techniques, and workload-specific parameters (e.g., batch size, input length), the framework systematically searches for configurations that deliver exceptional performance. Designed to address specific requirements and constraints, the framework enables researchers and practitioners to explore a vast optimization space, predict the performance of untested configurations, and make informed deployment decisions with confidence. By abstracting the optimization process into a modeling and search problem, the framework significantly reduces the time and effort traditionally required for deployment tuning, while ensuring scalability and adaptability across diverse hardware platforms and real-world scenarios.

Contributions. This work makes the following key contributions:

- We conduct systematic experiments that reveal key performance issues in LLM inference, including memory utilization drop-offs caused by framework-hardware mismatches and latency divergence under varying batch sizes and parallel configurations.
- We propose GUIDE, a novel deployment system, that integrates hardware modeling, model analysis, inference frameworks, deployment strategies, and optimization techniques to efficiently explore multi-dimensional optimization spaces.
- We validate the proposed simulator's effectiveness, achieving an error range of 25% to 55% across performance metrics, with an average error of 30% to 42%, effectively supporting deployment decisions in diverse configurations.

2 Background & Motivation

2.1 Transformer Models

Transformer models [9] have significantly advanced artificial intelligence by enabling breakthroughs in natural language processing (NLP), computer vision, and multimodal tasks. As shown in Figure 2, the architecture consists of stacked Transformer blocks, each composed of Multi-Head Attention (MHA), Feed-Forward Networks (FFN), and Normalization (Norm) layers. Residual connections facilitate gradient flow during training, while positional encodings provide sequence order information that the architecture itself lacks.

The Transformer architecture is divided into an encoder and a decoder. The encoder processes input sequences and builds contextual representations by capturing global dependencies through self-attention. The decoder generates output sequences by attending to both the encoder's output and its

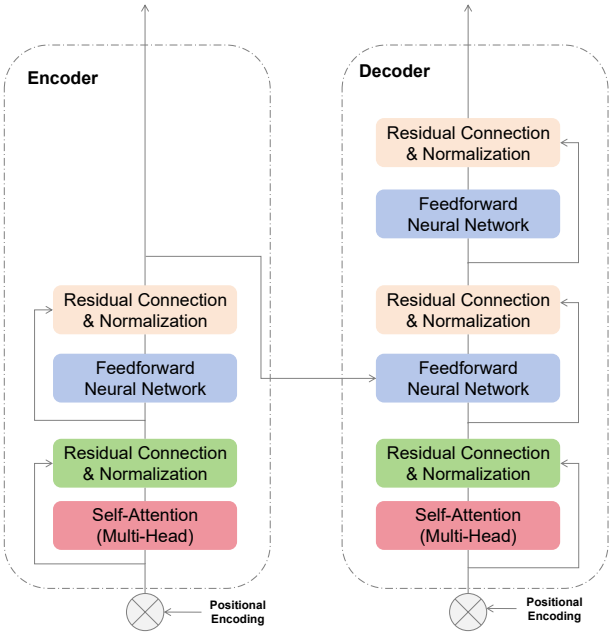

Figure 2: Basic transformer architecture.

own previously generated tokens. This design supports a wide range of sequence-to-sequence tasks, such as machine translation and text generation.

2.2 LLM Inference Challenges

Deploying Large Language Models (LLMs) presents several technical challenges, primarily arising from memory bottlenecks, computational complexity, and latency requirements.

A critical issue is memory usage during inference, dominated by the storage of Key-Value (KV) pairs in self-attention mechanisms. These KV pairs are retained across all layers, with memory consumption scaling linearly with sequence length and model depth. For long input sequences or deep models, this can quickly exceed hardware capacities, especially on GPUs with limited memory. Techniques like memory-efficient attention and model quantization have been proposed to alleviate this constraint, but they often involve trade-offs in precision or computational overhead.

Another major challenge is the quadratic computational complexity of the self-attention mechanism with respect to sequence length. This limits throughput, particularly for tasks requiring long context windows or high concurrency. Underutilization of hardware resources is common in such scenarios, further exacerbated by mismatches between model architectures and hardware capabilities.

Latency requirements add another layer of complexity, especially for real-time applications. Factors such as batch size, sequence length, and parallelism strategies heavily influence latencies. High-concurrency settings, small batch sizes, or workloads with variable input lengths often result in significant performance degradation due to suboptimal scheduling or increased communication overheads in distributed systems.

The heterogeneity of hardware platforms and inference frameworks further complicates deployment. Different hardware and software systems often exhibit inconsistencies in their ability to handle diverse workloads, requiring careful tuning to achieve optimal performance under practical scenarios. These challenges necessitate systematic optimization strategies to balance memory usage, throughput, and latency across heterogeneous environments.

2.3 LLM Inference Optimization

Optimizing the inference of LLMs requires addressing challenges such as memory bottlenecks, computational complexity, and latency constraints. Over the years, researchers have developed a variety of techniques, including algorithmic optimizations, deployment strategies, and specialized inference frameworks. This section provides an overview of these methods and their relevance to LLM inference.

2.3.1 Algorithmic Optimizations

Algorithmic techniques are fundamental to overcoming memory and computational challenges in LLM inference. Quantization methods, such as SmoothQuant [10] and LLM.int8 [11], reduce the precision of weights and activations, significantly lowering memory consumption while maintaining acceptable accuracy. Pruning approaches, such as SparseGPT [12], identify and remove redundant parameters, thereby reducing computational complexity and accelerating inference. FlashAttention optimizes the self-attention mechanism by minimizing memory access overheads, enabling efficient processing of long sequences. These techniques have demonstrated their effectiveness in addressing specific bottlenecks, making them widely adopted in real-world deployments.

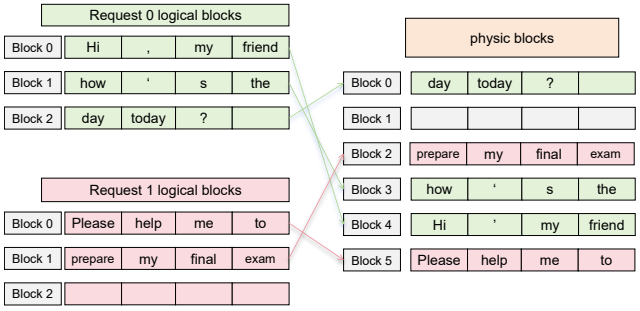

Figure 3: Logical-to-physical block mapping in vLLM.

2.3.2 Inference Frameworks

The growing demand for efficient LLM deployments has led to the emergence of numerous inference frameworks, each designed to address different aspects of the deployment process. Popular frameworks such as vLLM, DeepSpeed-FastGen [13], TGI (Text Generation Inference) [14], FasterTransformer [15], and LLaMA.cpp [16] represent diverse strategies for optimizing latency, throughput, and resource utilization. These frameworks provide tailored solutions for various deployment scenarios, ranging from high-performance cloud systems to resource-constrained edge environments.

vLLM introduces dynamic batching and parallelized token generation to improve computational efficiency and reduce latency. As shown in Figure 3, the figure illustrates the logical-to-physical block mapping framework used in vLLM. FastGen leverages scalability through dynamic splitting and fusion techniques integrated into its backend architecture. As illustrated in Figure 4, the DeepSpeed-FastGen backend consists of two main components: DeepSpeed-MII, which supports continuous batching and dynamic splitting, and DeepSpeed-Inference, which utilizes block KV-cache for efficient inference. TGI supports distributed inference with model sharding, enabling large-scale deployments across multi-GPU or multi-node clusters. FasterTransformer focuses on optimizing inference for NVIDIA hardware, leveraging advanced kernel-level optimizations. LLaMA.cpp, in contrast, caters to resource-constrained environments by employing aggressive quantization and memory management techniques.

Although this work primarily evaluates vLLM and FastGen due to their unique optimization strategies, the methodologies discussed are broadly applicable to other frameworks, reflecting the diversity of approaches in this area.

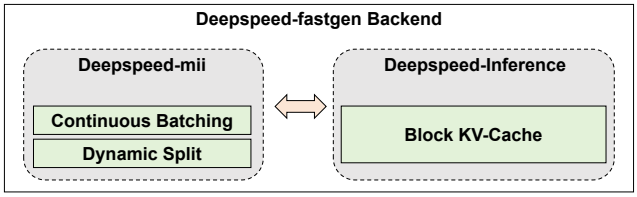

Figure 4: The architecture of the DeepSpeed-FastGen backend, showing continuous batching and dynamic splitting in DeepSpeed-MII, and block KV-cache in DeepSpeed-Inference.

2.3.3 Deployment Strategies

Deployment strategies, such as tensor parallelism, data parallelism, and pipeline parallelism, are critical for distributing computations across hardware resources. Tensor parallelism partitions model parameters across GPUs, enabling efficient handling of large-scale models. However, the approach introduces substantial communication overheads, as intermediate results must be synchronized during inference. Data parallelism, by splitting input data across devices, is well-suited for batch processing but requires careful synchronization to maintain consistency. Pipeline parallelism divides model layers across GPUs, allowing simultaneous execution of different stages of the model but introducing latency due to inter-stage dependencies.

2.4 Experimental Insights

2.4.1 Memory Utilization Drop-Offs

Both vLLM and FastGen exhibit sharp memory utilization drop-offs at critical points across specific hardware and model combinations. For vLLM, dramatic declines are observed on GPUs such as RTX 4090 and V100, where memory utilization drops abruptly as batch sizes increase from smaller values (e.g., 8) to larger ones (e.g., 32) for models like Qwen, OPT, and LLaMA (Figure 5. On A6000 GPUs, similar drop-offs occur with LLaMA, while FastGen, despite maintaining relatively higher utilization levels under most conditions, also encounters significant declines, such as with the OPT model on RTX 4090 and the LLaMA model on A6000. A100 GPUs exhibit more stable performance, yet inefficiencies persist when handling complex models like Qwen and OPT.

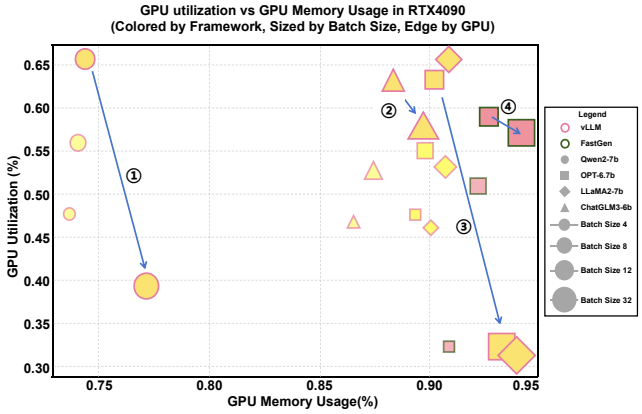

Figure 5: Memory utilization drop-offs.

These abrupt drop-offs stem from mismatches between KV-cache management strategies and hardware memory allocation mechanisms. vLLM's fixed-granularity KV-cache allocation strategy, while effective for small-batch tasks, leads to severe memory fragmentation and underutilization as batch sizes grow. On GPUs like A100, which allocate memory in large blocks (e.g., 2MB), these inefficiencies are magnified. FastGen, though employing a more adaptive caching mechanism, struggles with certain model and hardware combinations, where suboptimal allocation persists.

Moreover, the heterogeneous characteristics of models, such as KV-cache size and access patterns, further exacerbate these issues. Uniform caching strategies fail to adapt

to these variations, particularly as batch sizes increase, leading to sudden memory inefficiencies for both frameworks. These findings reveal not only the performance bottlenecks in memory management but also the significant optimization opportunities that exist for improving GPU resource utilization across diverse workloads and hardware configurations.

2.4.2 Batch Size and Latency Divergence

Latency performance shows a striking divergence between vLLM and FastGen as batch size increases, with a clear inflection point observed. For small-batch tasks (Batch Size ≤ 16), vLLM achieves significant latency advantages over FastGen, making it highly suitable for real-time applications. However, as batch size grows, vLLM's latency gradually deteriorates, and at Batch Size 32, FastGen surpasses vLLM, demonstrating stable and scalable performance. This divergence is particularly pronounced on high-end GPUs such as RTX 4090 and A6000, where vLLM's latency increases sharply, while FastGen maintains consistent performance over a wider range of batch sizes (Figure 6).

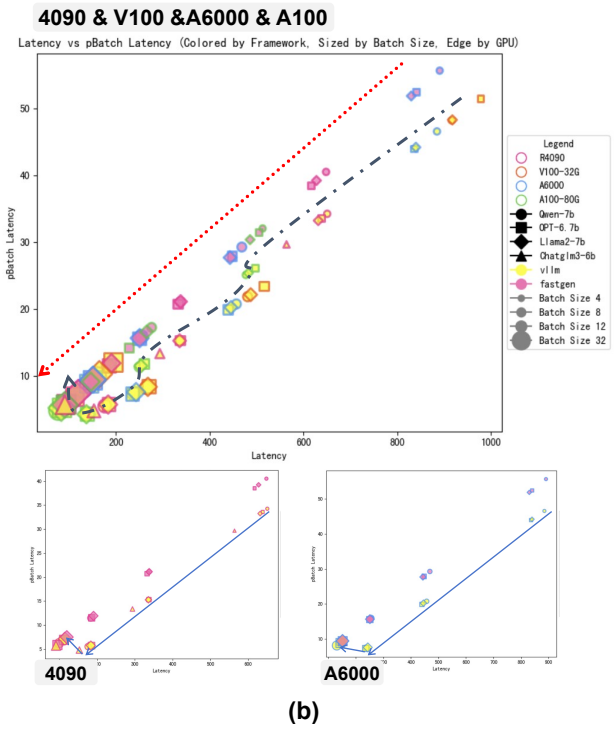

Figure 6: inference latency turn-off.

This divergence can be attributed to the fundamental differences in the resource allocation and task scheduling strategies employed by the two frameworks. vLLM prioritizes prompt processing by allocating substantial resources to KV-cache initialization and storage. While this approach performs well for small-batch tasks, it leads to significant resource contention as batch sizes grow. Specifically, vLLM's static KV-cache allocation strategy fails to scale with increasing workload complexity, causing delays in token generation and sharp latency spikes. In contrast, FastGen employs a dynamic splitting and fusion strategy that optimizes matrix operations by breaking them into smaller, parallelizable chunks. This approach minimizes contention, reuses intermediate results, and ensures balanced resource utilization, enabling FastGen to scale efficiently with larger batch sizes.

These findings highlight a significant optimization space for balancing latency performance across batch sizes. The trade-off between real-time performance for small batches and scalability for larger batches reveals opportunities to improve task scheduling and resource allocation strategies, particularly for high-end GPUs and diverse workload requirements.

2.4.3 Performance Under Parallel Environments

When tensor parallelism (TP) increased from TP=1 to TP=2, a divergence in first token latency (TTFT) was observed. For models like LLaMA, vLLM exhibited noticeable increases in TTFT, while FastGen maintained stable latency and even achieved slight reductions in some cases. As shown in Figure 7 and Figure 8, vLLM's TTFT rises with increasing parallelism, whereas FastGen demonstrates relatively consistent performance. Specifically, in the figures, the bottom-left corner represents the optimal performance, and the top-right corner indicates the worst performance. The green box highlights the performance with a single GPU, while the red line represents the performance with 2 GPUs. The blue arrow illustrates the trend from single GPU to multi-GPU, and the black line shows how this trend changes with increasing prompt length.

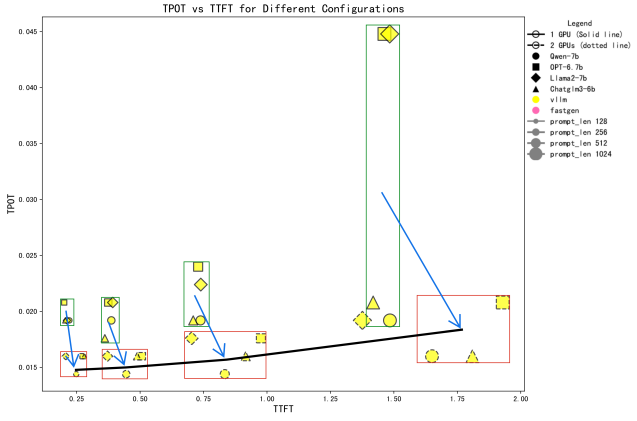

Figure 7: Latency Performance Under Parallel Environments for vllm.

The observed divergence is primarily caused by the complex interaction between tensor parallelism (TP) and the computational and communication patterns of different phases. During the Prefill phase, input prompts are processed in a

single pass, requiring intensive computation and frequent synchronization of intermediate results across GPUs. As parallelism increases, communication overhead grows significantly, leading to reduced Prefill efficiency. In the Decoding phase, where token generation proceeds step-by-step (each step depending on the previous one), cumulative synchronization delays further exacerbate bottlenecks in the generation process. These results suggest that TTFT divergence is not only influenced by hardware communication and parallel strategies but also by the intricate interaction between model architectures and computational phases.

These findings highlight that in tensor parallel environments, there is considerable room for optimization in the computational and communication patterns of different phases. By analyzing the key bottlenecks in Prefill and Decoding phases and understanding the factors affecting their efficiency, it is possible to explore better parallel strategy designs to improve inference performance.

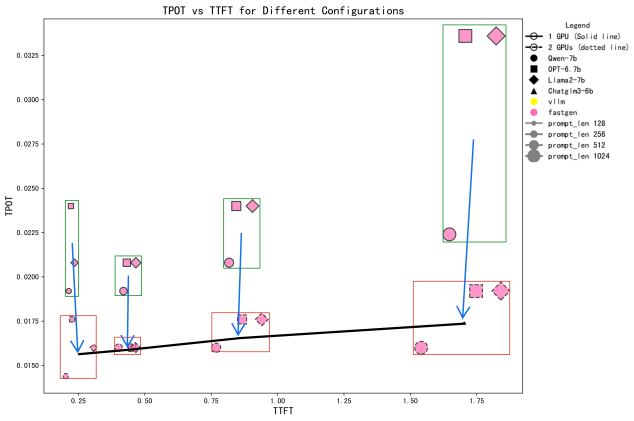

Figure 8: Latency Performance Under Parallel Environments for deepspeed-fastgen.

2.4.4 Dynamic Batch Size Optimization

Batch size selection plays a critical role in determining latency performance, as it directly affects how computational resources are utilized. Different tasks exhibit distinct optimal batch sizes due to their unique computational characteristics and resource demands. For tasks with high computational intensity, such as matrix multiplications with large dimensions, the GPU's computational units can quickly reach saturation even at smaller batch sizes. In contrast, tasks with lower computational demands scale efficiently to larger batch sizes before encountering hardware bottlenecks. Beyond these points, further increasing the batch size often leads to diminishing returns as resources such as compute units and memory bandwidth become fully utilized, limiting scalability and causing latency to increase.

This variation in optimal batch size highlights the intricate interaction between task characteristics and hardware constraints. Computationally intensive tasks saturate GPU resources at smaller batch sizes, leaving little room for further scalability, while less intensive tasks allow for greater scaling before bottlenecks arise. These differences make it clear that a one-size-fits-all batch size configuration is inherently suboptimal, as it fails to account for the varying demands of different workloads.

These observations reveal a significant optimization space for tailoring batch size configurations to specific task characteristics and hardware conditions. By analyzing the relationship between computational intensity, hardware utilization, and latency performance, it is possible to explore strategies that dynamically adjust batch size to better align with workload demands.

2.5 Motivation

The insights from our experiments highlight critical bottlenecks in LLM inference, revealing a vast optimization space shaped by the intricate interplay of multiple factors, including hardware configurations, model architectures, inference frameworks, deployment strategies, optimization methods, and workload parameters (e.g., batch size, sequence length). These factors, combined with the computational characteristics of the workloads, create a highly complex and sensitive environment where small changes in one dimension can significantly impact overall performance. Current methods, which often rely on static configurations or isolated tuning, fail to address these dynamic dependencies, leaving substantial optimization potential untapped in real-world deployments.

To address these challenges, we propose a systematic framework for modeling and simulating LLM inference. This framework aims to explore the multi-dimensional optimization space created by interactions among hardware, models, frameworks, and workload characteristics. By identifying the key factors that influence performance and understanding their interdependencies, our approach provides actionable insights for intelligent deployment decisions. It bridges the gap between theoretical advancements and practical requirements, paving the way for scalable, efficient, and deployable LLM inference across diverse configurations and scenarios.

3 Framework

To mitigate the performance bottlenecks identified in Section 2, we propose an intelligent deployment system designed to optimize inference performance for large-scale models. This system systematically models various configurations and simulates the performance across different inference setups, allowing for the identification of the optimal solution under complex inference configurations. It explores the multi-dimensional parameter space of model inference, including hardware platforms, inference frameworks, parallel strategies,

and optimization techniques. By integrating advanced modeling approaches with simulation-based optimization, the system effectively addresses challenges arising from complex factor interactions and dynamic bottlenecks, delivering robust and actionable performance improvements.

The intelligent deployment system is built around two core objectives. First, it aims to accurately model and predict the performance of large-scale models under diverse configurations by capturing the nuances of hardware, frameworks, and workload parameters. Second, it seeks to automate the optimization of deployment strategies, enabling users to achieve near-optimal performance with minimal manual intervention. A distinctive feature of this system is its adaptability to workload characteristics, hardware constraints, and framework-specific behaviors, ensuring consistent and reliable results across a wide range of deployment scenarios.

3.1 Overview

GUIDE addresses these challenges by automating the exploration of the multi-dimensional parameter space involved in large-scale model inference. It dynamically adapts to different hardware configurations, inference frameworks, and parallel execution strategies, ensuring that it identifies the most efficient deployment configurations for a given task. At its core, GUIDE optimizes both memory and computational resource usage, enabling the deployment of large-scale models with minimal manual effort while achieving maximized performance.

One of the key capabilities of GUIDE lies in its intelligent hybrid parallel simulation. By combining data parallelism (DP) and tensor parallelism (TP), it simulates various GPU configurations to determine the optimal parallel strategy. This simulation process minimizes total execution time in order to find the best parallel config. The system models the computational load and memory overhead of various stages in the inference process, including the prefill and decode phases. It dynamically adjusts batch sizes and sequence lengths based on these models to fit within the constraints of available GPU memory. This adjustment abstracts the influence of inference frameworks such as vllm and fastgen on the inference process, and to model these frameworks, it simulates the dynamic batch processing and dynamic split flow.

To further enhance its predictive capabilities, GUIDE incorporates the Roofline model [17], similar to the approach used by LLM-Viewer [18], which models the computational and memory overheads of inference. This integration allows it to analyze task performance and identify bottlenecks in computation and memory bandwidth, providing insights that guide decisions about parallel strategies and resource allocation. Coupled with simulation-based optimization, the system evaluates task execution time and throughput for different parallel configurations. Based on these metrics, it selects the top-performing configurations, ensuring that performance goals are met under the given constraints.

In addition to these analytical capabilities, GUIDE automates the generation of deployment configurations. By analyzing hardware and workload characteristics, it produces a set of potential configurations that can handle diverse deployment scenarios without requiring manual fine-tuning. This automation streamlines the process of adapting to new environments, making the system highly versatile and efficient.

Through these techniques, GUIDE empowers users to achieve near-optimal inference performance for large-scale models. Its adaptability ensures that resource utilization is maximized and task execution is completed in the shortest possible time. Whether deployed in single-GPU setups or large multi-GPU clusters, the system's flexibility and robustness make it suitable for a wide range of deployment environments.

As shown in Figure 9, GUIDE considers various factors and simulates the performance characteristics of the actual inference process, modeling the multi-dimensional parameter space involved in large-scale model inference. It dynamically adapts to different hardware configurations, inference frameworks, and parallel execution strategies, ensuring that it identifies the most efficient deployment configurations for a given task.

3.2 Inference Engine

Step 1: Memory Usage and Maximum Parallelism Calculation

To evaluate the memory overhead, which primarily arises from the key-value (kv) storage and model weight storage, we first perform a mathematical analysis using the Model Analyzer component. This step does not require the inference framework itself such as vllm or fastgen since they do not affect the total amount of memory consumption during prefill and decode phases, but takes into account user inputs for optimization such as FlashAttention and H2O which reduce the memory consumption. The analysis focuses on calculating the kv overhead for a batch size of 1. The kv overhead is derived by averaging the memory usage for a single token during both the prefill and decode phases.

The available GPU memory is then calculated as:

$$\text{Available Memory} = \text{GPU Memory} - \text{Model Slice Memory}$$

(where Model Slice Memory is determined by TP splitting).

Using this, the maximum parallelism can be derived as:

$$\text{Maximum Parallelism} = \text{DP} \times \left(\frac{\text{Available Memory}}{\text{kv overhead per request}} \right),$$

and the maximum allowed batch size (Max Batch Size) is:

$$\text{Max Batch Size} = \frac{\text{Maximum Parallelism}}{\text{Data Parallelism}}.$$

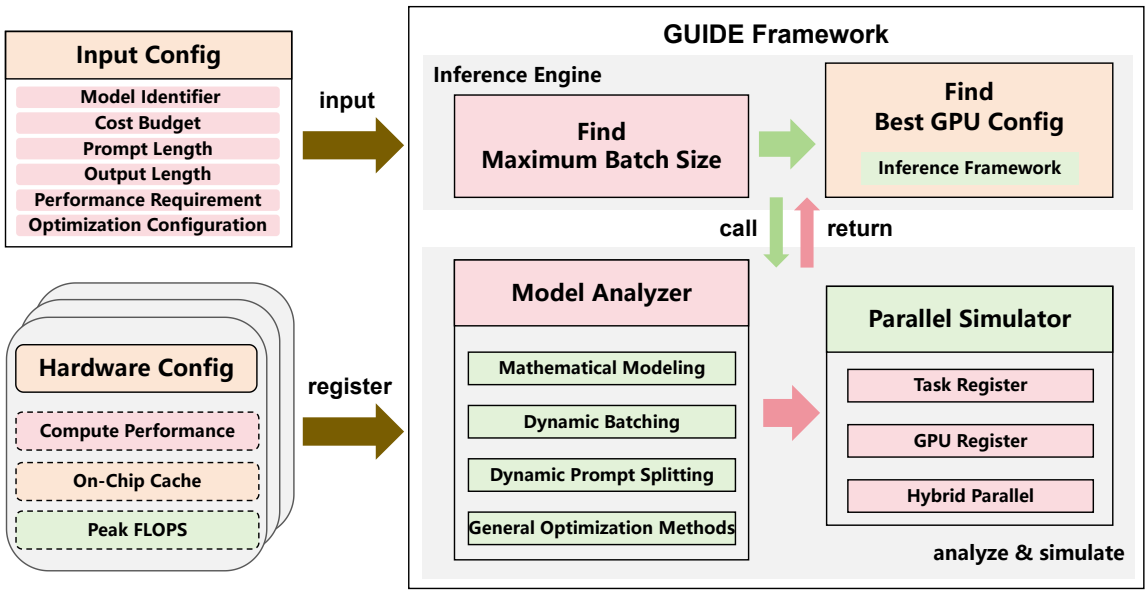

Figure 9: Overview of GUIDE. The figure shows the architecture of GUIDE, where task-specific input configurations are generated based on user-provided requirements, and hardware specifications are pre-registered into the system. The Inference Engine interacts with the Model Analyzer and Parallel Simulator to determine the maximum batch size and the best GPU configuration. The selection of the best GPU configuration also incorporates the choice of inference frameworks. Ultimately, GUIDE generates optimal configurations tailored to diverse workloads, including parameters such as batch size, GPU selection, inference frameworks, and deployment strategies.

This process is carried out using the *Parallel Simulator*, which helps identify the optimal parallel configuration (data parallelism, dp, and tensor parallelism, tp) based on the maximum batch size.

Step 2: Task Scheduling and Simulation for Optimal Performance

Once the maximum batch size has been determined, the next step is to analyze the performance of the chosen configuration using the Model Analyzer. In this step, we analyze both vllm and FastGen inference models under the same configs as step 1. Specifically, the kv overhead for each token during the prefill and decode phases is calculated. The total memory required for processing a request is the sum of the kv overhead and the model weight size.

The Model Analyzer also calculates the computational overhead based on the number of tokens to be generated. This information is then used to generate a task list for GUIDE, which consists of tasks for the prefill and decode stages. Each task includes both memory and computation costs. The task list is passed to the Parallel Simulator, which calculates the data transfer and memory read/write times caused by different parallel strategies. The simulator outputs the optimal parallel configuration, recording the top three best configurations.

GUIDE then evaluates the performance of different configurations based on inference time or throughput. For each configuration, the simulation results give the best parallel configuration and the corresponding inference time. And the final throughput is derived as follows:

$$\text{single\_gpu\_throughput} = \frac{1}{\text{TPOT}}$$

and

$$\text{multi\_gpu\_throughput} = \frac{N \cdot T_{\text{single}}}{1 + \log_2(P_{\text{TP}})},$$

where TPOT represents the token processing time per operation for a single GPU, N is the number of GPUs, T_{single} denotes the single-GPU throughput, and P_{TP} stands for the parameter parallelism.

TTFT for the prefill phase is simply the time per token for the prefill phase.

These results are used to identify the top three configurations with the best performance based on inference time or throughput, where the configuration with the lowest inference time and highest throughput is considered optimal.

3.3 Model Analyzer

In this section, we describe the second key component of GUIDE, which is the Model Analyzer. This technique focuses on optimizing memory utilization and computational efficiency by dynamically adjusting key parameters, such as batch size, sequence length, and KV cache size, based on the available GPU memory and model requirements. As shown in Figure 10, the Model Analyzer integrates dynamic batch

size adjustment, sequence length adjustment, and key-value (KV) cache optimization with H2O to simulate throughput while using these optimization and inference framework.

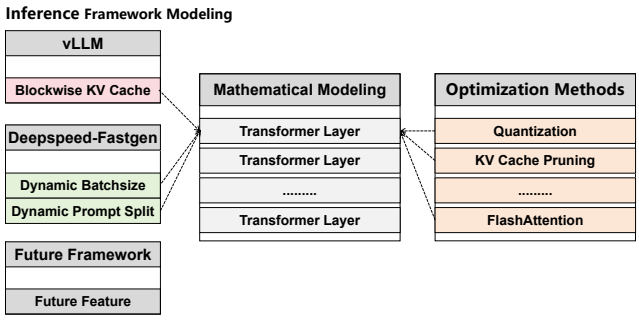

Figure 10: Overview of the Model Analyzer technique in GUIDE. The figure shows how inference framework modeling and optimization methods collaboratively act on each transformer layer to enable more accurate modeling and evaluation of LLM inference.

3.3.1 Dynamic Batch Size Adjustment

The dynamic batch size adjustment mechanism is designed to optimize the batch size while ensuring it adheres to the constraints imposed by GPU memory limitations. This approach consists of two primary steps: calculating the memory required per request and dynamically adjusting the batch size.

First, the memory required for the key-value (KV) storage per request is calculated as:

$$\text{kv\_byte\_per\_request} = 2 \times N \times H \times S \times B \times kv,$$

where: - N represents the number of hidden layers, - H is the hidden size of the model, - S is the sequence length, - B is the batch size, - kv denotes the memory in bytes required for each key-value pair.

To isolate the memory consumption independent of the batch size, the memory required for KV storage per request without the influence of batch size is expressed as:

$$\text{kv\_byte\_per\_request\_without\_batchsize} = 2 \times N \times H \times S \times kv.$$

Given the GPU's total memory capacity C_{gpu} and the memory occupied by the model C_{model}, the available memory for KV storage is:

$$C_{\text{available}} = C_{\text{gpu}} - C_{\text{model}}.$$

The maximum allowable batch size under these memory constraints is determined as:

$$B_{\text{max}} = \left\lfloor \frac{C_{\text{available}}}{\text{kv\_byte\_per\_request\_without\_batchsize}} \right\rfloor.$$

If the input batch size B_{input} exceeds this maximum allowable batch size, the batch size is adjusted to:

$$B = \min(B_{\text{input}}, B_{\text{max}}).$$

To further optimize memory usage and enhance prompt processing performance, the adjusted batch size is divided into smaller sub-batches. The sub-batch splitting process is designed to ensure that each sub-batch fits within the available memory while maintaining computational efficiency.

The size of each sub-batch is determined by a dynamically chosen parameter, denoted as split_size, which balances memory constraints with the need for efficient computation. Specifically, the batch size B is divided into sub-batches of size split_size, ensuring that the number of sub-batches is maximized without exceeding memory limitations. The number of sub-batches is determined by the ratio of the batch size to the split size, with any remaining portion of the batch forming an additional sub-batch.

3.3.2 Dynamic Sequence Length Adjustment

The core of the dynamic sequence length adjustment is to optimize the sequence length based on the GPU's available memory, ensuring that memory constraints are respected while maximizing efficiency. The process consists of two main steps: calculating the memory required per request and adjusting the sequence length dynamically.

First, the memory required for each request, without considering the sequence length, is calculated as:

$$\text{kv\_byte\_per\_request\_without\_seqlen} = 2 \times N \times H \times B \times kv,$$

Given the available memory, the maximum sequence length S_{max} that can fit within the memory constraints is calculated as:

$$S_{\text{max}} = \left\lfloor \frac{C_{\text{available}}}{\text{kv\_byte\_per\_request\_without\_seqlen}} \right\rfloor.$$

If the input sequence length exceeds the maximum sequence length allowed by the available GPU memory, the sequence length is adjusted to fit within the memory constraints.

Additionally, to simulate the impact of long prompts, the sequence is divided into smaller segments. The size of each segment is determined by a base value, which is then multiplied by a tuning factor to allow for fine-tuning. The total number of splits required is calculated by dividing the input sequence length by the split size and rounding up to the nearest integer.

Finally, the sequence length is adjusted based on the number of splits, with the adjusted sequence length being the

product of the number of splits and the split size. If the adjusted sequence length exceeds the original sequence length, the adjustment is applied to align the sequence length with the available memory. Otherwise, the sequence length remains unchanged.

The final adjusted sequence length is returned, which will either be the adjusted length if it does not exceed the original input length, or the original sequence length if no adjustment was necessary.

3.4 Parallel Simulator

In this section, we introduce the third key technique of GUIDE: the Parallel Simulator. This simulator focuses on optimizing task execution in multi-GPU environments by simulating hybrid parallelism, combining Data Parallelism (DP) and Tensor Parallelism (TP). The goal is to find the optimal parallel configuration that minimizes the total execution time of the tasks.

As shown in Figure 11, the Parallel Simulator handles the task execution flow by simulating both Data Parallelism and Tensor Parallelism to minimize the execution time across multiple GPUs.

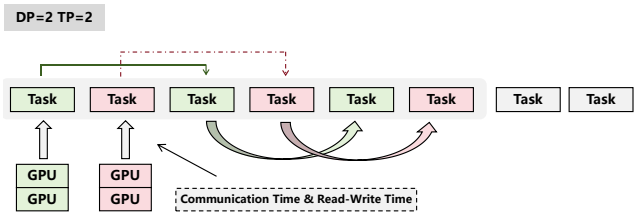

Figure 11: Parallel execution flow in the Parallel Simulator. The figure shows how tasks are distributed using DP and TP, with communication and read-write overheads modeled.

3.4.1 Task and GPU Models

The core of the simulator consists of two main models: the *Task* class and the *GPU* class.

- Task Model: Each task in the simulator is represented by a `Task` object, which holds the data size and the compute load of the task. - GPU Model: The `GPU` class captures the hardware characteristics of a single GPU, including its compute performance, memory bandwidth, communication bandwidth, and latency.

3.4.2 Roofline Model for Performance Analysis

To analyze whether a task's performance is constrained by compute power or memory bandwidth, the simulator uses the Roofline model from LLM-Viewer. This model plots the performance ceiling of a task based on the available compute and memory resources. Specifically, the Roofline model can be expressed as:

$$\text{Performance} = \min\left(\frac{\text{compute\_load}}{\text{compute\_time}}, \frac{\text{memory\_bandwidth}}{\text{data\_size}}\right)$$

Where: - compute_load is the number of floating point operations (FLOPs) required by the task. - compute_time is the time taken by the task on the GPU based on its compute performance (TFLOPS). - memory_bandwidth is the rate at which data can be transferred between memory and the processor.

The Roofline model helps to identify the bottleneck for each task—whether the task is memory-bound or compute-bound—and informs the simulation of execution time.

3.5 Task Execution and Hybrid Parallelism Modeling

3.5.1 Time Calculation

The task execution time can be broken down into three key components. The first component is the *compute time*, which is determined by the total computation load of the task divided by the computational power of the GPU. Mathematically, it is expressed as:

$$\text{compute\_time} = \frac{\text{compute\_load}}{\text{gpu\_compute\_power}}$$

The second component is the *data read/write overhead*, which refers to the time required to read and write data to and from the GPU memory. This is calculated by dividing the total data size by the GPU memory read/write bandwidth:

$$\text{data\_read\_write\_time} = \frac{\text{total\_data}}{\text{gpu\_memory\_bw}}$$

The third component is the *data transfer overhead*, which represents the time required to transfer data between system memory and the GPU memory. This is determined by the total amount of data transferred and the GPU memory communication bandwidth:

$$\text{data\_transfer\_time} = \frac{\text{total\_data\_transferred}}{\text{gpu\_memory\_comm\_bw}}$$

The total task execution time is then the sum of these three components.

3.5.2 Hybrid Parallelism (DP and TP) Modeling

Hybrid parallelism, which combines Data Parallelism (DP) and Tensor Parallelism (TP), plays a crucial role in optimizing task execution. In the context of hybrid parallelism, the task is divided into multiple groups, with each group of tasks processed in parallel across several GPUs. In Data Parallelism

(DP), the task is split into batches, with each batch processed by multiple GPUs in parallel. The execution time for each batch is determined by the slowest GPU in that batch, taking into account any communication delays between GPUs.

On the other hand, Tensor Parallelism (TP) involves splitting the model parameters themselves across multiple GPUs. Within each batch, each GPU processes a portion of the model's parameters, and all GPUs in the group work simultaneously on a given task. This strategy helps in reducing the computational load on each individual GPU and improves overall performance by leveraging the parallelism of the model's structure.

The hybrid parallelism approach combines these two techniques, enabling the simulation of a system where tasks are distributed across multiple GPUs with both data and tensor parallelism. This results in a more efficient utilization of the available resources and a reduction in the total execution time for large-scale tasks.

3.5.3 Simulating Hybrid Parallelism

A key feature of the simulator is its ability to simulate hybrid parallelism, which combines both Data Parallelism (DP) and Tensor Parallelism (TP). The `simulate_task_execution` function enables the simultaneous use of both parallel strategies. The function first generates all possible combinations of DP and TP configurations using the `get_configurations` method, ensuring that the number of GPUs used does not exceed the available resources.

The simulation process begins with configuration generation, where the `get_configurations` method creates all feasible DP and TP configurations by varying the number of GPUs allocated to the DP and TP tasks. The objective here is to explore the full range of possible configurations and identify the optimal balance between the two parallel strategies.

Once the configurations are generated, the simulator proceeds with the task execution simulation. The `simulate_hybrid_parallel` method is used to simulate the execution of tasks under each DP and TP configuration. This method considers important factors such as compute time, memory bandwidth, and communication delays. By accounting for these variables, the simulator provides a detailed assessment of how the tasks are executed in a hybrid parallel environment.

Finally, after simulating the execution for all configurations, the simulator evaluates the total execution time for each configuration. The optimal configuration is the one that minimizes the total execution time across all tasks. This configuration is selected based on its ability to best balance the execution times of both DP and TP tasks, while adhering to the constraints of available GPU resources. The goal is to achieve the most efficient use of resources, ensuring that the task execution is optimized across the hybrid parallel environment.

4 Implementation

In this section, we describe the implementation of our system, which is built using Python and consists of a frontend and a backend. The frontend serves as the user interface, while the backend handles core functionalities, including user request processing and performance analysis.

4.1 Backend Implementation

The backend, implemented in Python, acts as the core component for processing user requests and executing logic. It is composed of three main modules: 'inference engine', 'model analyzer', and 'parallel simulator'. The 'inference engine' module serves as the interface, accepting user inputs such as budget, selected model, generated sequence length, and input sequence length. Based on these inputs, it triggers backend processes to provide outputs, including the hardware configuration with the highest throughput, the configuration with the minimum inference time, and the top three hybrid parallelism configurations optimized for performance and budget constraints. To achieve these outputs, 'inference engine' coordinates with 'model analyzer' and 'parallel simulator', calculates performance metrics, and presents the results to the user for informed decision-making.

The second module, 'model analyzer', built on the 'llm-viewer' framework, evaluates the performance of inference frameworks and optimizations. This module integrates frameworks such as `vllm` and `fastgen` for model assessment and considers key-value (KV) store optimizations, including `h2o`, to enhance data handling during inference. By analyzing the impact of these frameworks and optimizations on throughput and inference time, 'model analyzer' identifies the most suitable configurations for the user's specific requirements.

The third module is a hybrid parallelism simulator designed to test various parallelization strategies within the constraints of the user's budget. It evaluates combinations of data parallelism, pipeline parallelism, and model parallelism to determine configurations that minimize inference time and maximize throughput. By considering the effect of budget constraints on these strategies, 'parallel simulator' identifies optimal configurations that balance performance with resource availability, enabling users to achieve efficient model deployment under limited resources.

4.2 User Interface

A web-based user interface (UI) was developed to configure simulation parameters and display results. The UI allows users to select a model from a dropdown menu, input a budget, and configure parameters such as sequence length, throughput requirement, latency preference, and precision tolerance.

Among these, sequence length, throughput requirement, and latency preference provide both predefined options for users unfamiliar with the parameters and custom input fields for users requiring precise control. Users can initiate the computation process, and the results, including relevant performance metrics, are displayed in an organized format. The interface layout is shown in Figure 12.

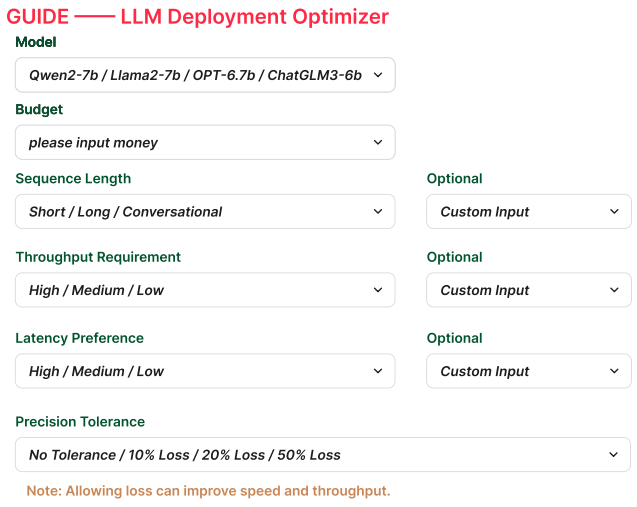

Figure 12: UI layout. The figure shows the user interface of GUIDE, where users can specify model, budget, sequence length, throughput, latency, and precision preferences, along with optional custom inputs.

5 Evaluation

5.1 Experimental Setup

We evaluate the accuracy and performance of our simulator by comparing simulated results with real experimental data under various configurations. The primary goal of the evaluation is to model and analyze the behavior of large language model (LLM) inference tasks in real-world environments, then compare it with the predictions of our simulator. The evaluation involves two parallel tracks: real-world execution on actual hardware and simulated execution within the modeling framework, with the results from both tracks used to compute prediction errors as key evaluation metrics.

In real-world experiments, tasks are designed to simulate common LLM inference scenarios. Each task involves a set of input prompts, where the prompt lengths are randomly generated with fixed values to represent realistic usage patterns. For the outputs, the simulated tasks generate fixed-length sequences, while in real-world tasks, we intentionally limit the maximum output length to align with the fixed output length in the simulation. This approach introduces some unavoidable discrepancies, as real-world model outputs are inherently variable and difficult to fully control.

Each model was tested with varying configurations to evaluate the simulator's accuracy under diverse scenarios. The configurations included batch sizes of 4, 8, 16, 32, which represent different levels of parallel workloads, and prompt lengths of 128, 256, 512, 1024, which reflect diverse input complexities. The output length was fixed at 256 tokens in the simulation environment for standardization, while in real-world experiments, the maximum output length was constrained to 256 tokens to align with the simulation. These configurations were chosen to comprehensively cover real-world usage patterns and provide a robust evaluation of the simulator's performance.

In the simulator, the same configurations, including hardware settings, batch sizes, prompt lengths, and model architectures, are replicated. The simulator generates predictions for three key metrics: Time-to-First-Token (TTFT), which measures the time required to produce the first token; Decode Throughput, defined as the throughput during the decoding phase, measured in tokens processed per second; and Batch Latency, representing the total time required to complete the processing of an entire batch. These predictions are compared against real-world measurements to calculate the error values, which serve as indicators of the simulator's accuracy.

The experiments are conducted on diverse hardware, software, and model configurations. For single-GPU experiments, we tested vLLM on NVIDIA A100 80G, V100 32G, A6000, and RTX 4090 GPUs, while FastGen was evaluated on NVIDIA A100 80G, A6000, and RTX 4090. For multi-GPU experiments, we used NVIDIA A100 80G, A6000, and RTX 4090 under tensor parallelism for both frameworks. The large language models assessed include THUDM/chatglm3-6b, Qwen/Qwen2-7B, facebook/opt-6.7b, and meta-llama/Llama-2-7b-hf.

5.2 Results Analysis

5.2.1 Single-GPU Evaluation

In our single-GPU experiments, we evaluated a variety of models to assess their error performance across different configurations. To simplify the data presentation, we report the mean error rates across all tested models, as summarized in Table 2. Specifically, the mean errors for Batch Latency, TTFT, and Decode Throughput are 33.04%, 33.31%, and 51.43%, respectively, for `vLLM`, and 32.74%, 41.43%, and 54.94% for `FastGen`. Figure 13 provides a detailed view of these errors across different GPU types, input lengths, and batch sizes. This aggregated presentation offers insight into the general trends and variations in prediction errors under diverse configurations.

The results presented in Table 2 and Figure 13 reveal the complexities in accurately predicting GPU performance across diverse configurations. While the overall trends show

Table 1: Comparison of Error Metrics on a Single GPU.

Error Metric	vLLM (%)	FastGen (%)
Batch Latency	33.04	32.74
TTFT	33.31	41.43
Decode Throughput	51.43	54.94

increasing errors with larger input lengths and batch sizes, the `nvidia_A100_80G` exhibits relatively higher errors compared to other GPUs. This phenomenon can be attributed to the unique architecture and scaling characteristics of the A100. Specifically, the high computational density and parallelism of the A100 may lead to greater sensitivity to workload imbalance or suboptimal utilization of its hardware features. For example, under extreme configurations such as very large batch sizes or input lengths, the model's assumptions about uniform resource usage may deviate significantly from the actual hardware behavior.

Additionally, unmodeled factors such as kernel-level behavior, synchronization overhead, and cache utilization can further exacerbate the discrepancies between predicted and actual performance. These issues are particularly pronounced in high-performance GPUs like the A100, where the complexity of the hardware amplifies even minor inefficiencies in workload execution. Overall, these findings highlight the inherent challenges of building precise performance models, especially for cutting-edge GPUs operating under diverse and demanding workloads.

5.2.2 Multi-GPU Evaluation

In our multi-GPU experiments, we evaluated the error performance of `vLLM` and `FastGen` across different configurations. To simplify the presentation, Table 2 reports the averaged error rates for each model, focusing on three core metrics: Batch Latency, TTFT, and Decode Throughput. For `vLLM`, the errors are 25.19%, 29.56%, and 25.00%, respectively, while `FastGen` shows errors of 28.06%, 31.86%, and 37.62%. A detailed breakdown of these errors is provided in Figure 13, which visualizes the trends across different GPU types, input lengths, and batch sizes. The results highlight considerable variability in errors under diverse conditions, offering insights into the performance trade-offs involved in multi-GPU setups.

The multi-GPU results in Table 2 and Figure 13 reveal several key factors influencing prediction errors. Firstly, errors generally decrease compared to single-GPU setups, reflecting better workload distribution and parallelism. How-

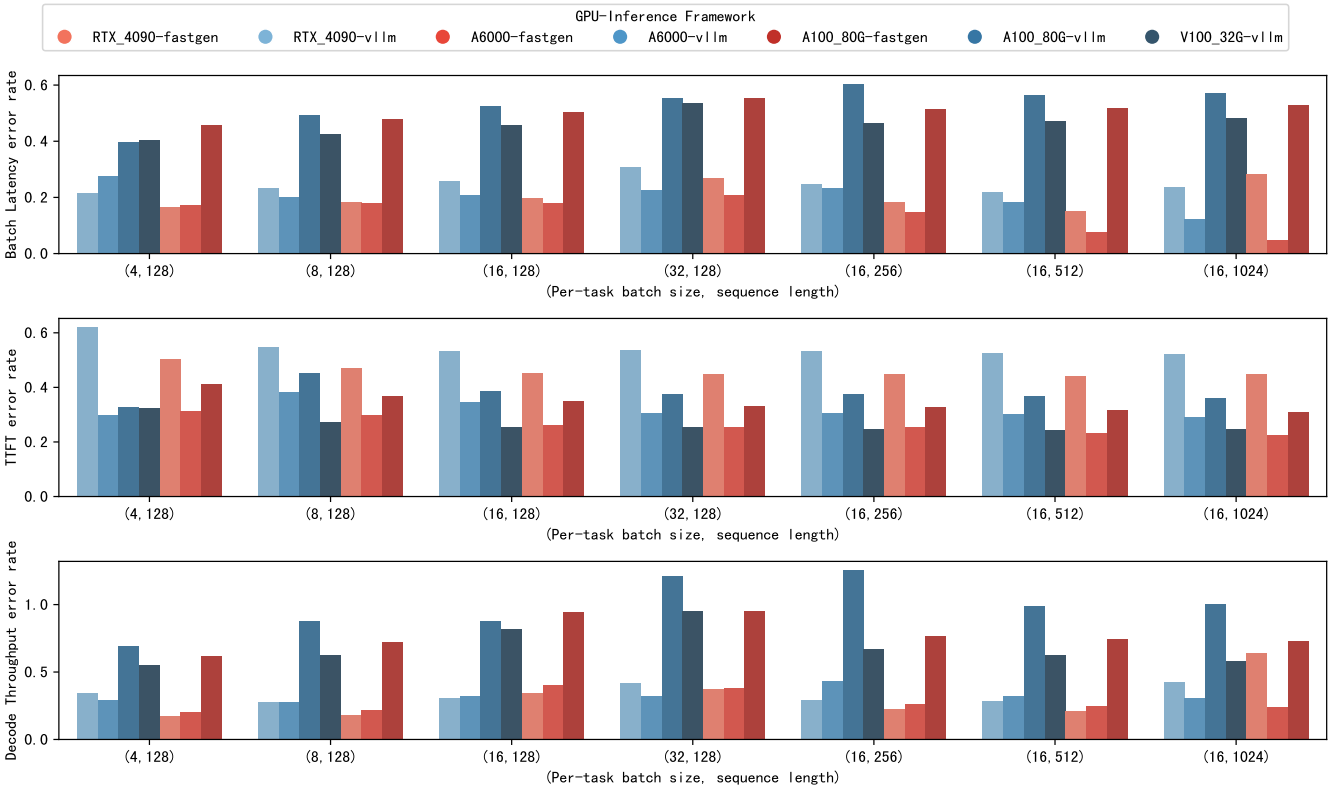

Figure 13: Comparison of performance metrics on a single GPU between vLLM and FastGen across different input lengths and batch sizes. The figure illustrates the batch latency, TTFT, and decode throughput for both models under identical conditions.

Table 2: Comparison of Error Metrics Across Multiple GPUs.

Error Metric	vLLM (%)	FastGen (%)
Batch Latency	25.19	28.06
TTFT	29.56	31.86
Decode Throughput	25.00	37.62

ever, as input lengths and batch sizes increase, errors begin to rise, driven by more complex inter-GPU communication and synchronization overheads. These overheads, while partially mitigated by modern GPUs like the `nvidia_A100_80G`, still contribute to deviations from predicted performance due to variable latency in data transfer and kernel synchronization.

In addition, the higher Decode Throughput errors observed for `FastGen` suggest greater sensitivity to scaling inefficiencies in multi-GPU environments. This could be linked to workload partitioning strategies, where imbalances in processing across GPUs lead to underutilization of computational resources or bottlenecks in specific GPUs. Furthermore, unmodeled system-level factors, such as memory contention and cache behavior, exacerbate these discrepancies, especially under extreme configurations.

6 limitations & Future Work

One of the main limitations of this study lies in the simulation accuracy. The models used for performance evaluation do not fully account for real-world conditions such as CPU-GPU interactions, power supply fluctuations, thermal dynamics, and the impact of other concurrently running applications. These factors, which significantly influence performance in practical scenarios, are difficult to simulate accurately and are not included in our analysis. As a result, the performance results may differ from what would be observed in a real-world environment, and the conclusions drawn from these simulations may not perfectly represent actual performance under varied operational conditions.

Another limitation stems from the abstraction used in modeling the inference framework. While the framework captures many important aspects of task scheduling and resource allocation, it is not a comprehensive representation of every component and interaction involved in a real inference system. Key factors such as hardware-level optimizations, architectural differences between GPUs, and the dynamic behavior of various software layers are simplified or omitted. This partial modeling inevitably introduces errors and discrepancies between the simulated and real-world performance, particularly in complex, heterogeneous environments.

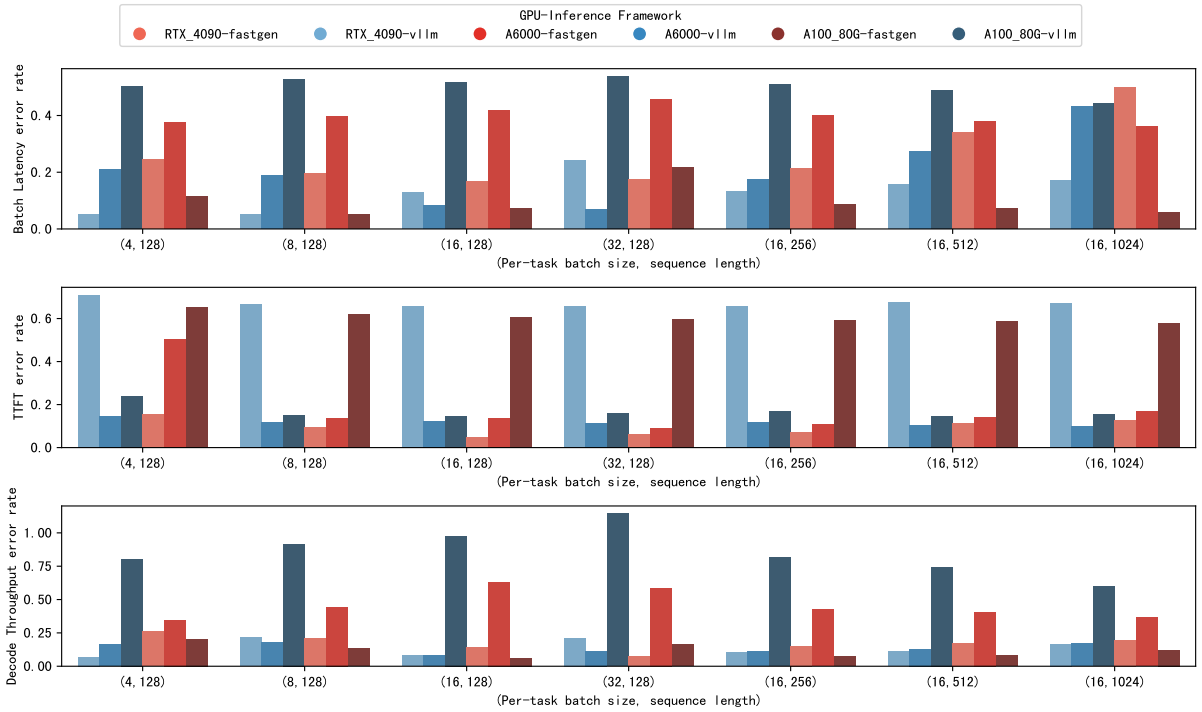

Figure 14: Comparison of performance metrics on multiple GPUs between VLLM and FastGen across different input lengths and batch sizes. The figure illustrates the batch latency, TTFT, and decode throughput for both models under identical conditions in a multi-GPU setup.

7 Related Work

7.1 Mathematical Modeling and Simulator-Based Performance Evaluation

Mathematical modeling and simulation frameworks are crucial for optimizing large language model (LLM) inference systems. Recent advancements focus on system-level efficiency and user-centric performance metrics. Andes [19] improves Quality-of-Experience (QoE) in text streaming by dynamically allocating GPU resources, achieving up to 3.2× QoE gains under high loads. Etalon critiques traditional metrics like throughput and latency, introducing the fluidity index to better capture real-time performance.

Simulation tools such as Vidur [20] and GenZ [21] provide valuable insights into platform requirements. Vidur combines experimental profiling with predictive modeling, maintaining less than 9

Optimization techniques also improve resource utilization. Nanoflow [22] uses nano-batching and operation-level pipelines to overlap memory, compute, and network operations, achieving up to 1.91× throughput improvements. Uellm [23] combines resource profiling, batch scheduling, and dynamic deployment strategies to reduce latency and enhance GPU utilization, demonstrating up to 4.98× throughput gains.

Despite these advancements, many tools focus on specific aspects or deployment scenarios. Vidur and Etalon primarily address static profiling and metric refinement, which may overlook dynamic, real-world workloads. Metrics like "idealized runtime" enable cross-model comparisons but abstract away critical workload variability and system contention factors. These limitations highlight the need for comprehensive frameworks that integrate user-centric metrics, real-time adaptability, and cost-aware optimization strategies.

7.2 Optimization Strategies for Large-Scale Models

Optimizing inference for large-scale language models (LLMs) focuses on memory efficiency, throughput, and latency. Several strategies target different parts of the inference pipeline.

Memory management is crucial for long-context generation. Infinigen [24] and H2O [25] improve Key-Value (KV) cache management. Infinigen speculates essential KV entries, reducing fetch overhead and improving offloading-based systems by up to 3×. H2O treats KV eviction dynamically, improving throughput by up to 29×. However, both depend on specific hardware optimizations, limiting generalizability.

Algorithmic innovations like vllm's PagedAttention and Razorattention [26] boost memory efficiency. Vllm reduces KV cache waste using virtual memory, achieving 2–4× throughput improvements for longer sequences. Razorattention compresses the cache by over 70

For computational efficiency, Deepspeed-fastgen and vllm offer significant gains. Deepspeed-fastgen improves throughput by 2.3× and reduces tail latency with dynamic prompt generation. Vllm's IO-aware tiling algorithm cuts memory reads/writes, achieving 3× speedup for GPT-2 [27] on long sequences.

Sarathi-Serve [28] optimizes throughput and latency by refining the prefill and decode phases. Its chunked-prefills and stall-free scheduling improve serving capacity by up to 5.6× for models like Falcon-180B [29], making it suitable for real-time applications.

7.3 Pre-trained Models for Large-Scale Inference

Pre-trained large language models (LLMs) are crucial in AI, with advancements in scalability, accessibility, and fine-tuning. The Qwen2 series, ranging from 0.5 to 72 billion parameters, excels in benchmarks like CMMLU [30] and HumanEval [31], supports 30 languages, and promotes community-driven fine-tuning. The Llama2 series, with models like llama2-Chat, performs well in dialogue tasks, offering an open alternative to proprietary models.

The chatglm family, especially GLM-4, competes with proprietary models like GPT-4 [32], adding tool integration for complex tasks, which emphasizes alignment and multimodal integration, especially for Chinese applications. The OPT initiative, with models from 125M to 175B parameters, focuses on reproducibility and sustainability, achieving GPT-3-level performance while reducing carbon footprints.

Despite these advancements, challenges remain in balancing scalability, efficiency, and alignment. Models like Qwen2 and GLM-4 require substantial computational resources, limiting access for smaller organizations. Improving alignment across languages and cultures is an ongoing research area, with future work focusing on resource-efficient training and stronger alignment techniques to increase accessibility and societal impact.

8 Conclusion

In conclusion, GUIDE provides a robust and scalable framework for optimizing the inference performance of large-scale language models. By systematically addressing critical bottlenecks identified through experimentation, such as memory inefficiencies, latency divergence, and multi-GPU scaling issues, GUIDE bridges the gap between theoretical model performance and practical deployment requirements. Its integration of dynamic modeling, simulation-based optimization, and intelligent deployment strategies equips researchers and practitioners with practical tools to navigate the complexities of real-world environments. With further advancements in its capabilities and enhanced predictive accuracy, GUIDE has the potential to become an indispensable resource for deploying

large-scale models efficiently, cost-effectively, and adaptively across heterogeneous environments.

References

[1] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[2] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models, 2022.

[3] 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.

[4] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, et al. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024.

[5] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[6] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, 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*, pages 1–15, 2021.

[7] 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 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[8] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference. *arXiv preprint arXiv:2401.08671*, 2024.

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

[10] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

[11] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[12] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023.

[13] Connor Holmes, Masahiro Tanaka, Michael Wyatt, Ammar Ahmad Awan, Jeff Rasley, Samyam Rajbhandari, Reza Yazdani Aminabadi, Heyang Qin, Arash Bakhtiari, Lev Kurilenko, et al. Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference, 2024. *URL https://arxiv.org/abs/2401.08671*.

[14] Hugging Face. Text generation inference. https://github.com/huggingface/text-generation-inference, 2023.

[15] NVIDIA. Fastertransformer: A library for high performance inference of transformer models. https://github.com/NVIDIA/FasterTransformer, 2023.

[16] Georgi Gerganov. llama.cpp. https://github.com/ggerganov/llama.cpp, 2023.

[17] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[18] Zhihang Yuan, Yuzhang Shang, Yang Zhou, Zhen Dong, Zhe Zhou, Chenhao Xue, Bingzhe Wu, Zhikai Li, Qingyi Gu, Yong Jae Lee, et al. Llm inference unveiled: Survey and roofline model insights. *arXiv preprint arXiv:2402.16363*, 2024.

[19] Jiachen Liu, Zhiyu Wu, Jae-Won Chung, Fan Lai, Myungjin Lee, and Mosharaf Chowdhury. Andes: Defining and enhancing quality-of-experience in llm-based text streaming services. *arXiv preprint arXiv:2404.16283*, 2024.

[20] Amey Agrawal, Nitin Kedia, Jayashree Mohan, Ashish Panwar, Nipun Kwatra, Bhargav Gulavani, Ramachandran Ramjee, and Alexey Tumanov. Vidur: A large-scale

simulation framework for llm inference. *Proceedings of Machine Learning and Systems*, 6:351–366, 2024.

[21] Abhimanyu Bambhaniya, Ritik Raj, Geonhwa Jeong, Souvik Kundu, Sudarshan Srinivasan, Midhilesh Elavazhagan, Madhu Kumar, and Tushar Krishna. Demystifying platform requirements for diverse llm inference use cases. *arXiv preprint arXiv:2406.01698*, 2024.

[22] Kan Zhu, Yilong Zhao, Liangyu Zhao, Gefei Zuo, Yile Gu, Dedong Xie, Yufei Gao, Qinyu Xu, Tian Tang, Zihao Ye, et al. Nanoflow: Towards optimal large language model serving throughput. *arXiv preprint arXiv:2408.12757*, 2024.

[23] Yiyuan He, Minxian Xu, Jingfeng Wu, Wanyi Zheng, Kejiang Ye, and Chengzhong Xu. Uellm: A unified and efficient approach for llm inference serving. *arXiv preprint arXiv:2409.14961*, 2024.

[24] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. {InfiniGen}: Efficient generative inference of large language models with dynamic {KV} cache management. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 155–172, 2024.

[25] Zhenyu Zhang, Ying Sheng, Tianyi Zhou, Tianlong Chen, Lianmin Zheng, Ruisi Cai, Zhao Song, Yuandong Tian, Christopher Ré, Clark Barrett, et al. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *Advances in Neural Information Processing Systems*, 36:34661–34710, 2023.

[26] Hanlin Tang, Yang Lin, Jing Lin, Qingsen Han, Shikuan Hong, Yiwu Yao, and Gongyi Wang. Razorattention: Efficient kv cache compression through retrieval heads. *arXiv preprint arXiv:2407.15891*, 2024.

[27] 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.

[28] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming throughput-latency tradeoff in llm inference with sarathi-serve. *arXiv preprint arXiv:2403.02310*, 2024.

[29] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, et al. The falcon series of open language models. *arXiv preprint arXiv:2311.16867*, 2023.

[30] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[31] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[32] 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.