FLEXLLM: Flexible and Cost-Efficient LLM Serving with Heterogeneous GPUs

Kihyun Kim[1], Jinwoo Kim[1], James J. Kim[2], Dong Li[3], Youngjae Kim[1,†]

[1]Sogang University, Seoul, Republic of Korea, [2]Soteria Inc., [3]University of California, Merced, CA, USA

{kion777, jinwookim, youkim}@sogang.ac.kr, jkim@soteria-sys.com, dli35@ucmerced.edu

Abstract—The autoregressive nature of LLMs causes memory bottlenecks, requiring multi-GPU parallelization. Prior work has mainly optimized strategies for homogeneous setups, overlooking heterogeneous configurations and service-level objectives (SLOs) despite growing GPU cost-performance gaps. This paper introduces FLEXLLM, a framework that predicts execution times and selects cost-efficient strategies in heterogeneous GPU environments while satisfying latency-per-token (LPT) SLO constraints. The proposed system (i) bridges theoretical predictions and actual performance through a Linear Correction Function (LCF) and (ii) performs SLO-aware cost-efficiency optimization based on human reading speeds (≤150 ms per token). Our evaluation demonstrates that FLEXLLM identifies cost-efficient configurations meeting SLO requirements, achieving significant cost reductions compared to performance-oriented approaches. Heterogeneous GPU analysis reveals that SLO-aware parallelization strategy selection yields up to 2.28× cost-efficiency differences, demonstrating that architecture selection under SLO constraints is as critical as hardware investment.

Index Terms—Heterogeneous GPU Computing, Large Language Model Serving, Performance Prediction, Cost-Efficiency Optimization, Parallelization Strategies

I. INTRODUCTION

Large language models (LLMs) based on the Transformer architecture have revolutionized natural language processing, delivering remarkable performance across a wide range of applications, including machine translation, text summarization, and conversational AI [1]. Despite their impressive capabilities, deploying these models in real-world production environments poses substantial economic and technical challenges. In particular, the autoregressive nature of LLMs results in growing memory demands that scale with both input sequence length and batch size. These memory requirements frequently exceed the capacity of a single GPU, making it necessary to adopt multi-GPU parallelization strategies to enable efficient inference and maintain acceptable response times (§II-A).

Various parallelization strategies have been proposed to accelerate LLM inference, including Model Parallelism (MP), Attention Offloading (AO), and Data Parallelism (DP) [2]–[4] (§II-B). Each of these strategies provides unique advantages depending on the model structure and hardware capabilities. However, most prior research has concentrated on optimizing each parallelization method in isolation, often under the assumption of homogeneous GPU environments. These studies primarily aim to maximize raw performance without considering the broader trade-offs among cost, hardware heterogeneity, and practical deployment constraints faced in real-world scenarios. Therefore, existing studies overlook

a critical challenge in selecting the optimal parallelization strategy for given workloads and hardware configurations.

This problem has become increasingly complex due to two converging trends. **First**, the GPU market is experiencing growing cost-performance disparities—although upgrading from an NVIDIA A100 to an H100 can yield 2–3× performance gains, the associated cost increase often surpasses this improvement by a significant margin [5]. **Second**, many LLM-based applications operate under relatively low throughput requirements that do not fully utilize the available GPU resources, rendering the use of high-end GPUs economically inefficient in such contexts [6].

These factors have driven interest in heterogeneous GPU configurations that strategically combine high-performance GPUs (H-GPUs) and cost-effective GPUs (L-GPUs). Under the token-based pricing models prevalent in the industry (e.g., cost per million tokens) [7], [8], optimizing hardware costs directly impacts service competitiveness. However, *no existing framework systematically predicts execution times across different parallelization strategies in heterogeneous environments* or provides practical guidelines for optimal architecture selection.

In this paper, we present FLEXLLM, a mathematical model-based solver that systematically identifies the optimal parallelization strategy among MP, AO, and DP for heterogeneous GPU environments, minimizing deployment costs while satisfying latency-per-token (LPT) service-level objective (SLO). The solver employs performance modeling that accurately estimates execution times for the three major parallelization strategies through a Linear Correction Function (LCF) that adjusts theoretical FLOPS-based predictions using empirical measurements, achieving high accuracy with minimal profiling overhead. Furthermore, through comprehensive model-driven simulation analysis, FLEXLLM offers practical guidelines for selecting economically optimal architectures tailored to specific hardware configurations and user requirements.

Based on evaluations with the OPT-1.3B and OPT-2.7B models on real heterogeneous GPU setups, the practical deployment guidelines suggested by FLEXLLM demonstrate up to 2× improvements in tokens-per-dollar efficiency. It also demonstrates the ability to select the most cost-efficient GPU configuration and parallelization strategy while satisfying the SLO constraint.

II. BACKGROUND

A. Large Language Model Inference and Memory Bottleneck

Transformer-based large language models (LLMs) follow an autoregressive inference approach, generating one token at a

---

† Y. Kim is the corresponding author.

time conditioned on all previously generated tokens [9]. Each Transformer layer is composed of a self-attention mechanism and a feed-forward network (FFN). While the self-attention component captures inter-token dependencies, the FFN applies nonlinear transformations. This sequential generation process can be divided into two computationally distinct phases: the *prefill* phase and the *decode* phase.

The prefill phase processes the entire input sequence in parallel to produce the first output token and builds the Key-Value (KV) cache. This phase exhibits high parallelism and arithmetic intensity, characterizing it as *compute-bound* [10].

In contrast, the decode phase generates tokens autoregressively using cached Key and Value vectors. As sequence length increases, KV cache access grows linearly while computation per step remains small, making it *memory-bound* [10].

The memory requirement for the KV cache, denoted as M_{KV}, increases linearly with the sequence length S, batch size B, number of layers L, model dimension D, and numerical precision in bytes:

$$M_{KV} = 2B \cdot S \cdot L \cdot D \cdot \text{precision}(bytes) \quad (1)$$

In practical serving scenarios, this linear growth quickly exceeds the memory capacity of a single GPU. For example, serving the OPT-30B model with a batch size of 64 and sequence length of 8192 requires approximately 700 GB of memory for the KV cache alone [11]. Using the formula with OPT-30B parameters ($L = 48$, $D = 7168$), this yields $M_{KV} = 2 \times 64 \times 8192 \times 48 \times 7168 \times 2 \approx 672$ GB in FP16 precision. Therefore, large-scale inference necessitates multi-GPU configurations and well-designed parallelization strategies [12].

B. Parallelization Strategies for Heterogeneous Setup

To address memory constraints while leveraging heterogeneous GPU configurations, three major parallelization strategies have been proposed. Each exhibits distinct characteristics in terms of memory efficiency, implementation complexity, and communication overhead.

Model Parallelism (MP): Model parallelism alleviates memory pressure on a single GPU by distributing the computational workload of LLMs across multiple GPUs. Tensor Parallelism [13] partitions attention heads or feed-forward layers along tensor dimensions, while Pipeline Parallelism [2] divides the model into sequential layers and assigns them to different GPUs. However, these strategies often require that the number of GPUs align with the model architecture, which can limit scalability and flexibility [14]. In heterogeneous environments, MP can leverage different GPU capabilities by distributing computational loads according to hardware capacity, potentially improving resource utilization. However, MP incurs communication overhead due to activation tensor exchanges between GPUs at each layer boundary.

Attention Offloading (AO): Attention offloading is specifically designed for heterogeneous GPU environments, unlike traditional MP and DP strategies that assume homogeneous hardware configurations. AO emerged to exploit the distinct characteristics of different GPU types by delegating attention computations and KV cache storage to low-end GPUs (L-GPUs), while high-end GPUs (H-GPUs) handle compute-intensive operations such as feed-forward layers [3].

This architecture leverages heterogeneous hardware by creating specialized role separation that improves cost-efficiency over homogeneous configurations. However, AO can incur higher communication overhead compared to MP, as it requires transferring entire KV cache data between GPUs at each decoding step, whereas MP only needs to exchange intermediate tensor computation results across layer boundaries.

Data Parallelism (DP): Data parallelism splits the input batch across multiple GPUs, where each GPU independently executes inference using the full model. In heterogeneous environments, the batch is divided proportionally to each GPU's compute capability, with H-GPUs handling larger segments and L-GPUs handling smaller ones [13]. This strategy operates fully in parallel without requiring synchronization or intermediate data transfer between GPUs during inference. It offers low implementation complexity and minimal communication overhead, making it highly practical. However, since the entire model must be loaded on each GPU, memory usage increases across all participating GPUs. Additionally, if batch workload distribution is not properly balanced according to each GPU's processing capability, the slower device becomes a straggler that bounds the overall throughput, creating performance bottlenecks in heterogeneous configurations.

C. Related Work on Heterogeneous LLM Serving

A variety of research efforts [15]–[18] have focused on improving the efficiency of LLM serving in heterogeneous GPU environments.

System-Level Optimization Approaches: Jiang et al. [15] optimized cost-efficient serving plans for heterogeneous GPUs under budget constraints. Helix [16] modeled serving in heterogeneous clusters as a maximum-flow problem, while HexGen [17] supported inference in distributed heterogeneous systems via asymmetric parallelism and advanced scheduling strategies. SageServe [18] introduced an adaptive control mechanism for SLA-aware workloads.

Performance Prediction and Modeling: FNOPerf [19] proposed a machine learning–based model to predict LLM training time, while Imai et al. [20] introduced a sophisticated inference latency prediction model that combines the Roofline model with learning-based techniques, achieving high accuracy in single-GPU environments.

Limitations of Existing Approaches: While system-level optimization approaches have addressed heterogeneous resource allocation and scheduling, they typically focus on specific parallelization strategies without systematic comparison across Model Parallelism, Attention Offloading, and Data Parallelism under unified SLO-aware cost criterion. Performance prediction approaches are primarily limited to homogeneous settings and single-GPU environments. However, in real-world LLM applications where required throughput is often below maximum capacity, optimizing for cost-efficiency becomes more critical than fully utilizing high-end GPUs [6]. This fundamental shift in optimization objectives reveals several key limitations in current approaches:

(1) Lack of systematic performance models for predicting execution time across different parallelization architectures without extensive profiling; (2) Absence of quantitative frameworks for architecture-aware cost-efficiency evaluation that consider both performance and cost trade-offs; (3) Limited

practical guidelines for parallelization strategy selection based on hardware configurations and cost constraints.

III. PROBLEM DEFINITION

A. Architecture Selection Optimization

In multi-GPU LLM serving systems constrained by memory capacity, the core objective is to select the parallelization architecture (DP, AO, or MP) that satisfies the service-level objective (SLO) while minimizing cost. Unlike prior work that optimizes within a single strategy, our FLEXLLM framework evaluates architectures *across* strategies using an SLO-constrained cost-efficiency metric, defined as the total number of output tokens served per unit cost under acceptable latency per token. This enables us to identify the architecture with the lowest cost that satisfies a given latency constraint.

B. SLO Evaluation Metric: Latency per Token (LPT)

To quantitatively evaluate inference performance across diverse scenarios, we adopt the **Latency per Token (LPT)** as the primary service-level objective (SLO) metric. LPT represents the average time (in milliseconds) required to generate a single output token, and is defined as follows:

$$\text{LPT} = \frac{1000 \cdot T_{\text{E2E}}}{S_{\text{O}}} \quad [\text{ms/token}] \quad (2)$$

Here, T_{E2E} denotes the end-to-end batch inference time in seconds, and S_{O} is the number of output tokens. An architecture satisfies the SLO if it ensures $\text{LPT} \leq \text{LPT}_{\text{target}}$.

C. Limitation: Naive Cost-Efficiency Metric

The conventional cost-efficiency metric is defined as:

$$\text{CE}_{\text{naive}}(s) = \frac{3600 \cdot B \cdot S_{\text{O}}}{T_{\text{E2E}}(s) \cdot c_s} \quad [\text{tokens/\$}] \quad (3)$$

where B is the batch size, S_{O} is the number of output tokens per request, $T_{\text{E2E}}(s)$ denotes the end-to-end inference latency (in seconds) under architecture s, and c_s represents the hourly cost (in \$/h) of that configuration.

While this ratio aims to optimize cost per output token, it neither enforces a latency SLO nor penalizes *absolute cost*. Specifically, when SLO requirements are already satisfied, CE_{naive} continues to favor more expensive configurations that provide only marginal latency improvements at disproportionately higher costs. As a result, it presents the following limitations:

- *Absolute-cost insensitivity.* Because CE_{naive} optimizes a ratio, it may favor over-provisioned configurations that are much more expensive but provide only marginal improvements in tokens-per-dollar, even when SLO targets are already satisfied.

To address these limitations, we adopt an SLO-aware selection policy that prioritizes minimum absolute cost among configurations satisfying SLO requirements.

D. SLO-Aware Cost-Efficient Architecture Selection

We propose a two-stage selection strategy to identify the most cost-efficient architecture from a set of candidates $\mathcal{S} = \{\text{DP}, \text{AO}, \text{MP}\}$, under the constraint that the latency service-level objective (SLO) is satisfied.

a) Primary Criterion: Minimize Cost under Latency SLO:

$$s^* = \arg\min_{s \in \mathcal{S}} c_s \quad \text{subject to} \quad \text{LPT}(s) \leq \text{LPT}_{\text{target}} \quad (4)$$

b) Secondary Criterion: Maximize Cost-Efficiency as Tiebreaker: If multiple architectures satisfy the primary criterion with identical costs, we select the one with the highest token throughput per dollar, thereby maximizing overall cost-efficiency. This strategy ensures the selection of an architecture that not only meets the latency requirement ($\text{LPT}_{\text{target}}$) but also minimizes the actual deployment cost (c_s). In the case of cost-equivalent candidates, the policy favors configurations that deliver better utilization of budgeted resources.

This SLO-aware approach addresses the limitations of naive cost-efficiency metrics and enables cost-driven architecture selection for practical deployment scenarios.

IV. FLEXLLM SYSTEM DESIGN

In this section, we detail the overall system architecture and key design decisions of FLEXLLM, which automatically selects the optimal parallelization strategy based on the previously defined SLO-aware cost-efficiency policy. FLEXLLM is designed as a decision-support tool for cost-efficient LLM serving in heterogeneous GPU environments, centered around mathematical performance modeling and automated architecture selection.

A. System Overview and Architecture

The FLEXLLM framework comprises three main components: the **Input Processing Module**, **Performance Modeling Engine**, **Architecture Selection Optimizer**. This modular design enables independent development and extensibility of each component, allowing the system to flexibly adapt to various GPU configurations and workload specifications.

The system follows a hierarchical architecture, with each module serving a distinct role as outlined below:

Input Processing Module: This module receives the user-defined workload specifications and hardware configurations, and converts them into a normalized internal format.

Performance Modeling Engine: This component contains the core mathematical kernels for predicting the performance of each parallelization strategy. It includes a FLOPs Estimator, a Memory Bandwidth Analyzer, and a Linear Calibration Model, which together translate theoretical resource demands into realistic performance estimates.

Architecture Selection Optimizer: Based on the SLO-aware selection policy, this optimizer determines the most suitable architecture. It evaluates the performance of Model Parallelism, Attention Offloading, and Data Parallelism independently through dedicated evaluators, and generates detailed recommendations along with configuration guidelines.

The overall system workflow proceeds as follows: (1) Normalize and validate user inputs; (2) Predict performance for each architecture using the modeling engine; (3) Select the optimal architecture based on the SLO-aware cost-efficiency criterion; (4) Generate comprehensive recommendations and deployment configurations.

TABLE I
DEFINITION OF VARIABLES IN THE SYSTEM MODEL

Category	Variable Description
Workload	B: Batch size B_H, B_L: Sub-batch sizes for H-/L-GPU *(DP only;* $B_H+B_L=B$) S_I: Input sequence length S_O: Number of output tokens per request $\bar{S}_C$: Average context length ($= S_I + S_O/2$) S: Total sequence length ($= S_I + S_O$)
Model	L: Number of Transformer layers L_H: Number of layers placed on the H-GPU *(MP only;* $0 \leq L_H \leq L$) L_L: Number of layers placed on the L-GPU *(MP only;* $0 \leq L_L \leq L$, $L_H+L_L=L$) D: Model hidden dimension size D_{ffn}: Feed-forward network dimension b_e: Bytes per element (typically 2 bytes for FP16) M_W: Total model parameter size in bytes, M_W = Number of Params $\times b_e$ M_{KV}: Total KV cache size ($2BSLDb_e$) $S_{act, prefill}$: Activation tensor size during prefill phase (BS_IDb_e) $S_{act, decode}$: Activation tensor size during decode phase (BDb_e) S_{comm}: Communication volume during attention offloading (Q, K, V, and output Z) ($= 4BDb_e$) C_{off}: Offloading ratio for AO strategy ($0 \leq C_{off} \leq 1$)
Hardware	$C_{H/L}^{peak}$: Theoretical peak FLOPS of H/L-GPU $C_{H/L}^{eff}$: Effective FLOPS of H/L-GPU ($= C_{peak} \cdot \eta$) $\eta^{H/L}$: Efficiency factor of H/L-GPU, $\in [0.1, 0.9]$ BW_{inter}: Inter-GPU bandwidth (e.g., PCIe, NVLink)

B. Input Processing Module

1) Workload Specification Interface: The input processing module of FLEXLLM systematically collects workload characteristics and service-level requirements from the user. The system accepts user-specified parameters including the target LLM model (e.g., OPT-1.3B, OPT-2.7B), batch size B, input sequence length S_I, and output sequence length S_O that define the inference scenario. Based on the selected model, the system automatically extracts architectural parameters such as number of layers L, hidden dimension D, and feed-forward network dimension D_{ffn} from standard model configuration files. These core parameters are summarized along with their mathematical definitions in Table I. All collected inputs are normalized into a unified internal representation and passed to the performance modeling engine.

2) Hardware Configuration Collection: The system automatically detects the underlying heterogeneous GPU configuration consisting of a high-end GPU (H-GPU) and a low-end GPU (L-GPU) or accepts user-provided hardware specifications. For each GPU type, the system gathers key characteristics required for LLM inference time prediction, including manufacturer-rated peak FLOPS (C^{peak}) and effective FLOPS (C^{eff}) measured via a short GEMM microbenchmark over a small sweep of matrix sizes representative of our workload and data types, memory capacity (M), and memory bandwidth (BW), as summarized in Table I. Inter-GPU communication parameters such as communication bandwidth (BW_{inter}) and base latency (L_{comm}) are also collected to support accurate

TABLE II
TRANSFORMER LAYER FLOPS

Component	Prefill (S_I)	Decode ($\bar{S}_C$)
Self-Attention		
QKV Proj.	$6BS_ID^2$	$6BD^2$
Attention	$4BS_I^2D$	$4B\bar{S}_CD$
Output Proj.	$2BS_ID^2$	$2BD^2$
FFN ($D_{ffn} = 4D$)		
Up-Proj.	$8BS_ID^2$	$8BD^2$
Down-Proj.	$8BS_ID^2$	$8BD^2$
Total	$BS_I(24D^2 + 4S_ID)$	$B(24D^2 + 4\bar{S}_CD)$

latency estimation.

The hardware information collection module relies on system APIs (e.g., nvidia-smi) and vendor specification databases to retrieve these values. By preprocessing and validating all relevant hardware parameters upfront, the inference time prediction model can produce reliable estimates across diverse GPU environments.

C. Performance Modeling Engine

1) Theoretical Performance Calculator: The performance modeling engine implements mathematical performance models for the three heterogeneous architectures: Model Parallelism, Attention Offloading, and Data Parallelism. For each architecture, the engine decomposes the runtime of both the prefill and decode stages into fine-grained components and predicts the total inference latency accordingly. In addition, a measurement-driven linear correction model is applied to correct estimation errors in both stages (details are discussed in the following subsection).

2) Workload and Layer-wise FLOPs Modeling: The computational cost of a Transformer layer differs between the prefill and decode stages. Based on prior studies [21] and the model specifications listed in Table I, we analytically define the number of floating point operations (FLOPs) required for each stage as a function of the input sequence length (S_I), output sequence length (S_O), hidden dimension (D), and batch size (B). The detailed breakdown is provided in Table II. Using this decomposition, the per-layer FLOPs are defined as:

$$F_{layer}^{prefill} = BS_I(24D^2 + 4S_ID) \tag{5}$$

$$F_{layer}^{decode} = B(24D^2 + 4\bar{S}_CD) \tag{6}$$

Here, $\bar{S}_C$ represents the average context length, i.e., the number of previously generated tokens referenced during each decode step.

3) Architecture-specific Performance Models: Based on the FLOPs models above, we construct architecture-specific performance models to predict the total inference time under each of the three heterogeneous strategies.

Model Parallelism Performance Model: Each layer of the model is partitioned across H-GPUs and L-GPUs according to their memory capacity, and computations are executed sequentially. For both the prefill and decode stages, the compute time is calculated based on the effective FLOPS of each GPU (C_H^{eff}, C_L^{eff}). Additionally, communication overhead due to activation transfers across the interconnect is considered.

The prefill time is defined as:

$$T_{\mathrm{MP}}^{\mathrm{prefill}} = \frac{L_H \cdot F_{\mathrm{layer}}^{\mathrm{prefill}}}{C_H^{\mathrm{eff}}} + \frac{L_L \cdot F_{\mathrm{layer}}^{\mathrm{prefill}}}{C_L^{\mathrm{eff}}} + \frac{S_{\mathrm{act,\ prefill}}}{BW_{\mathrm{inter}}} \quad (7)$$

The decode time is calculated as:

$$T_{\mathrm{MP}}^{\mathrm{decode}} = \frac{L_H \cdot F_{\mathrm{layer}}^{\mathrm{decode}}}{C_H^{\mathrm{eff}}} + \frac{L_L \cdot F_{\mathrm{layer}}^{\mathrm{decode}}}{C_L^{\mathrm{eff}}} + \frac{S_{\mathrm{act,\ decode}}}{BW_{\mathrm{inter}}} \quad (8)$$

After applying the Linear Correction Function (LCF) to both stages, the total inference latency is given by:

$$T_{\mathrm{MP}}^{\mathrm{total}} = LCF\left(T_{\mathrm{MP}}^{\mathrm{prefill}}\right) + (S_O - 1) \cdot LCF\left(T_{\mathrm{MP}}^{\mathrm{decode}}\right) \quad (9)$$

Attention Offloading Performance Model: In the Attention Offloading architecture, the H-GPU is responsible for executing the entire inference pipeline, while the L-GPU partially offloads KV cache storage and a portion of the attention computations based on the offloading ratio C_{off}.

a) Prefill Stage: The H-GPU performs all layer-wise computations, and concurrently transfers a C_{off} fraction of the total KV cache to the L-GPU. Since computation and transmission can be overlapped, the total prefill latency is determined by the longer of the two operations:

$$T_{\mathrm{AO}}^{\mathrm{prefill}} = \max\left(\frac{L \cdot F_{\mathrm{layer}}^{\mathrm{prefill}}}{C_H^{\mathrm{eff}}}, \frac{C_{\mathrm{off}} \cdot M_{\mathrm{KV}}}{BW_{\mathrm{inter}}}\right) \quad (10)$$

b) Decode Stage: The H-GPU transmits a C_{off} fraction of the newly generated token's query (Q), key (K), and value (V) to the L-GPU during each decode step. The L-GPU performs the attention computation on the received Q, K, and V, and returns the result to the H-GPU. The remaining attention computation and the subsequent feed-forward operations are executed entirely on the H-GPU.

$$T_{\mathrm{AO}}^{\mathrm{decode}} = L\left[\frac{F_{\mathrm{FFN}}^{\mathrm{decode}} + F_{\mathrm{proj}}^{\mathrm{decode}} + (1 - C_{\mathrm{off}})\, F_{\mathrm{attn}}^{\mathrm{decode}}}{C_H^{\mathrm{eff}}} \right. $$
$$\left. + C_{\mathrm{off}}\left(\frac{F_{\mathrm{attn}}^{\mathrm{decode}}}{C_L^{\mathrm{eff}}} + \frac{S_{\mathrm{comm}}}{BW_{\mathrm{inter}}}\right)\right] \quad (11)$$

where $F_{\mathrm{proj}}^{\mathrm{decode}}$ denotes the per-layer FLOPs for QKV and output projections, $F_{\mathrm{FFN}}^{\mathrm{decode}}$ denotes the per-layer FFN FLOPs, and $F_{\mathrm{attn}}^{\mathrm{decode}}$ denotes the per-layer attention FLOPs (excluding projections).

c) Total Inference Time: The total inference latency under Attention Offloading is computed by applying the Linear Correction Function (LCF) to both the prefill and decode stage latencies:

$$T_{\mathrm{AO}}^{\mathrm{total}} = LCF\left(T_{\mathrm{AO}}^{\mathrm{prefill}}\right) + (S_O - 1) \cdot LCF\left(T_{\mathrm{AO}}^{\mathrm{decode}}\right) \quad (12)$$

Data Parallelism Performance Model: The input batch is split into two sub-batches, B_H and B_L, assigned to the H-GPU and L-GPU, respectively. Each GPU independently executes the full model, including both the prefill and decode stages. In the following equations, $F_{\mathrm{layer}}^{\mathrm{prefill}}$ and $F_{\mathrm{layer}}^{\mathrm{decode}}$ are evaluated using the respective GPU's batch size. Since both GPUs operate in parallel, the execution time for each stage is determined by the slower of the two.

The prefill time on each GPU is calculated as:

$$T_{\mathrm{DP},H}^{\mathrm{prefill}} = \frac{L \cdot F_{\mathrm{layer}}^{\mathrm{prefill}}}{C_H^{\mathrm{eff}}} \qquad T_{\mathrm{DP},L}^{\mathrm{prefill}} = \frac{L \cdot F_{\mathrm{layer}}^{\mathrm{prefill}}}{C_L^{\mathrm{eff}}} \quad (13)$$

Similarly, decode time is defined as:

$$T_{\mathrm{DP},H}^{\mathrm{decode}} = \frac{L \cdot F_{\mathrm{layer}}^{\mathrm{decode}}}{C_H^{\mathrm{eff}}} \qquad T_{\mathrm{DP},L}^{\mathrm{decode}} = \frac{L \cdot F_{\mathrm{layer}}^{\mathrm{decode}}}{C_L^{\mathrm{eff}}} \quad (14)$$

After applying the LCF correction to both stages:

$$T_{\mathrm{DP}}^{\mathrm{prefill}} = \max\left(LCF(T_{\mathrm{DP},H}^{\mathrm{prefill}}), LCF(T_{\mathrm{DP},L}^{\mathrm{prefill}})\right) \quad (15)$$

$$T_{\mathrm{DP}}^{\mathrm{decode}} = \max\left(LCF(T_{\mathrm{DP},H}^{\mathrm{decode}}), LCF(T_{\mathrm{DP},L}^{\mathrm{decode}})\right) \quad (16)$$

Therefore, the total inference time is given by:

$$T_{\mathrm{DP}}^{\mathrm{total}} = T_{\mathrm{DP}}^{\mathrm{prefill}} + (S_O - 1) \cdot T_{\mathrm{DP}}^{\mathrm{decode}} \quad (17)$$

Here, $B_H + B_L = B$, and by default, the batch is divided in proportion to GPU performance, though manual adjustment is possible depending on specific deployment scenarios.

4) Linear Correction Model: To bridge the gap between theoretical predictions and real-world latency measurements, we introduce a linear correction model for both the prefill and decode stages. Latency for each phase tends to increase linearly with the computational workload, and our model captures this trend to improve estimation accuracy. This linearity is justified by the following observations: (1) GPU matrix operations, when fully parallelized, scale linearly with FLOPs; (2) Transformer layers consist predominantly of large-scale matrix multiplications (attention and FFN); (3) Beyond the throughput saturation point, latency scales linearly with batch size.

$$LCF = \gamma_{\mathrm{arch}}^{\mathrm{phase}} \cdot (T_{\mathrm{arch}}^{\mathrm{phase}}) + \beta_{\mathrm{arch}}^{\mathrm{phase}} \quad (18)$$

where $\gamma_{\mathrm{arch}}^{\mathrm{phase}}$ is the scaling factor specific to each phase-architecture combination and $\beta_{\mathrm{arch}}^{\mathrm{phase}}$ represents the corresponding fixed overhead term, both empirically estimated from measurements.

Fig. 1 illustrates that for various input lengths and batch sizes, latency for both prefill and decode stages increases linearly with batch size across all architectures. This consistent trend supports the general applicability of the LCF-based correction model. Empirical cross-validation shows that the mean absolute percentage error (MAPE) remains below 5–20% across all ranges, demonstrating the practicality and accuracy of our linear correction model (see Section VI-B for details).

D. Architecture Selection Optimizer

The architecture selection process is based on the SLO and cost-efficiency criteria defined in Section III-B and Section III-D.

Given a user-specified workload, this module utilizes the precomputed inference time estimates for each architecture and performs the following two-step evaluation:

Step 1: SLO Satisfaction Check: For a fixed batch size, the module calculates the Latency per Token (LPT) for each candidate architecture and determines whether it satisfies the SLO constraint.
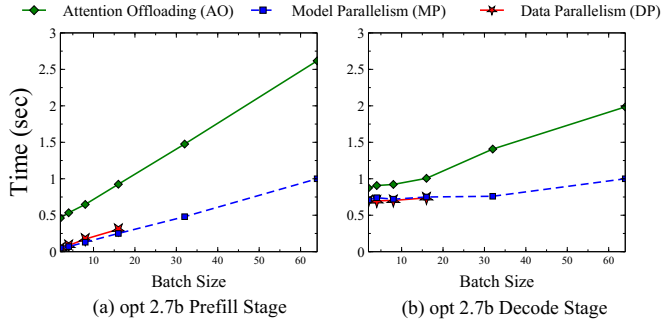
Fig. 1. In the OPT-2.7B model, we observe the runtime behavior of both the prefill and decode stages under varying batch sizes. As the batch size increases, the runtime exhibits a clear linear growth pattern in both stages. Due to GPU Out-of-Memory (OOM) errors beyond batch size 32, Data Parallelism measurements were limited to batch sizes up to 16.

Step 2: Minimum Cost Selection: Among the architectures that meet the SLO requirement, the module first identifies those with the minimum absolute cost. If multiple architectures have identical minimum costs, it selects the one with the highest cost-efficiency.

Finally, the optimizer recommends the architecture that minimizes deployment cost while satisfying the given workload requirements and SLO constraints.

V. IMPLEMENTATION

We developed FLEXLLM using Python (3.10.14) as a decision-support solver that systematically identifies the most cost-efficient parallelization strategy based on user-specified workload and hardware parameters. For the mathematical performance modeling and linear correction functions, we conducted numerical analysis using the NumPy library (1.24.3) and SciPy library (1.10.1).

For validating the parallelization strategies in FLEXLLM, we developed implementations supporting the three major approaches using PyTorch (2.7) and the FlexGen framework [21]. These implementations serve as ground-truth systems for evaluating the accuracy of FLEXLLM's mathematical models and architecture recommendations. Model Parallelism divides Transformer layers across two GPUs using PyTorch's DistributedDataParallel and performs computation by exchanging activation tensors through the NCCL backend. Attention Offloading transfers Query and KV cache using `cudaMemcpyAsync()` to perform attention operations on the L-GPU. Data Parallelism is implemented using PyTorch Distributed to split input batches and enable each GPU to perform inference independently.

VI. EVALUATION

A. Experimental setup

To validate the performance of FLEXLLM and analyze its cost-efficiency across diverse heterogeneous configurations, this study employed two distinct approaches: empirical validation experiments and model-based simulation experiments. The empirical experiments were conducted on actual workstations to quantitatively verify the model's prediction accuracy. The simulation experiments analyzed throughput and cost-efficiency trends based on controlled FLOPS performance ratios to simulate heterogeneous GPU characteristics.

Validation Setup: The validation experiment was conducted to evaluate the accuracy of the Linear Correction Function (LCF) calibration and the inference time prediction model in a homogeneous GPU environment. All experiments were performed on a workstation equipped with two NVIDIA GeForce RTX 2080 SUPER GPUs connected via a PCIe 1.0×8 interface ($\sim$ 2.0 GB/s). Both GPUs were locked to a fixed clock frequency of 2100 MHz to maintain a 1:1 FLOPS performance ratio. The detailed hardware configuration is summarized in Table III.

Correction and Validation Procedure: In the correction phase, latency measurements were collected on the OPT-1.3B and OPT-2.7B models under a small workload ($S_I = 128$, $S_O = 32$) by increasing the batch size from 2 up to the memory limit in powers of two. These measurements were used to derive the parameters (γ, β) of the Linear Correction Function (LCF). In the validation phase, the derived correction function was applied to Medium ($S_I = 512$, $S_O = 64$) and Large ($S_I = 1024$, $S_O = 128$) workloads to predict latency, and the predicted values were compared against actual measurements.

The parallelization strategies were implemented as follows: Model Parallelism (MP) splits Transformer layers evenly across two GPUs and exchanges activation tensors between them. Attention Offloading (AO) offloads the entire attention computation and KV cache to one GPU while the other GPU performs the FFN operations. Data Parallelism (DP) divides the input batch equally (1:1) across two GPUs, with each executing independently.

For each architecture and batch size, prefill and decode latency were measured five times, and the average was reported. CUDA cache clearing and GPU warm-up steps were applied prior to measurement. Prediction accuracy was evaluated using Mean Absolute Percentage Error (MAPE).

Simulation Setup: This simulation study applies the validated FLOPS-based latency prediction model to analyze throughput and cost-efficiency across heterogeneous GPU configurations. We normalized the FLOPS of the reference GPU (RTX 2080) to 1.0 and defined three virtual GPU pairings—H–H (1:1.0), H–L (1:0.5), and H–L (1:0.1)—to represent high-end and low-end ratios in today's GPU market. All GPUs are assumed to have 8 GB of memory, with hourly costs set to $6.00 for H–GPU, $3.00 for L–GPU (0.5× FLOPS), and $0.60 for L–GPU (0.1× FLOPS). For each GPU pairing, model, and batch size combination, we calculate the predicted latency using these normalized FLOPS ratios, derive throughput, and compute tokens per dollar (cost efficiency). Human reading comprehension speeds are typically 4–6 tokens/s [22].

Model	Type	Metric	Prefill	Decode
OPT-1.3B	DP	Med. MAPE (%) Lg. MAPE (%) γ β	6.23 18.62 0.336 0.00045	9.47 10.40 0.00905 0.01457
	MP	Med. MAPE (%) Lg. MAPE (%) γ β	5.17 11.69 0.6437 0.00287	3.84 6.42 0.4389 0.01546
	AO	Med. MAPE (%) Lg. MAPE (%) γ β	2.73 4.32 0.9326 0.45114	5.76 10.82 0.95533 0.01787
OPT-2.7B	DP	Med. MAPE (%) Lg. MAPE (%) γ β	6.58 14.55 0.29039 0.01391	3.16 17.20 0.00460 0.02208
	MP	Med. MAPE (%) Lg. MAPE (%) γ β	3.69 7.66 0.48493 0.00069	7.55 20.08 0.61392 0.02208
	AO	Med. MAPE (%) Lg. MAPE (%) γ β	3.41 0.87 1.06984 0.38430	8.92 11.61 1.11116 0.02641

Med./Lg.: Medium/Large workloads, γ and β are the phase-specific and architecture-specific correction parameters for the Linear Correction Function (LCF).

Accordingly, we set the target Latency Per Token (LPT) to 150 ms to ensure real-time interactive performance.

The results are visualized as a heatmap in Fig. 2, providing practical guidance for selecting an inference architecture and configuring GPU clusters.

B. Model Prediction Accuracy Validation

Table IV presents the validation results, showing MAPE values and corresponding correction parameters for Medium and Large workloads across different model sizes and parallelization strategies. For performance evaluation, we classified accuracy levels as excellent (below 5%), good (5-15%), acceptable (15-20%), and concerning (above 20%). Overall, the results demonstrate varying prediction accuracy depending on specific combinations of model architectures and parallelization strategies, with most configurations achieving performance within the acceptable range.

Analysis reveals distinct prediction performance characteristics across different parallelization architectures. Attention Offloading (AO) exhibited the most stable prediction performance overall. For Medium workloads, prefill stages achieve excellent accuracy (2.7-3.4% MAPE) while decode stages show good performance (5.8-8.9% MAPE). This superior performance stems from AO's inherent characteristics where communication dominates over computation, resulting in relatively predictable patterns. Model Parallelism (MP) showed moderate prediction accuracy but exhibited relatively high error rates, reaching the concerning threshold (20.08%) for decode steps in OPT-2.7B Large workloads. Data Parallelism (DP) demonstrated variable performance depending on configuration, with notably high MAPE values recorded in some Large workload prefill stages.

These variations can be attributed to architectural complexities: for MP, the intricate communication patterns and synchronization overhead resulting from model parameter partitioning across GPUs are not adequately captured by simple FLOPS-based prediction models. For DP, uneven memory
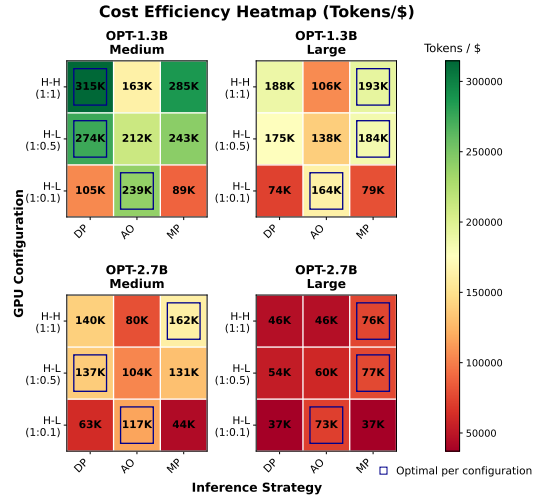

Fig. 2. Cost–efficiency across GPU configurations (rows) and inference strategies (columns) for four model–workload settings. Inner-Rectangles denote the optimal strategy for each GPU configuration. Values are in tokens per dollar.

access patterns during batch distribution processing and synchronization delays caused by throughput differences between GPUs contribute to increased prediction errors.

Model size analysis reveals that OPT-2.7B exhibits relatively higher prediction errors compared to OPT-1.3B. This can be attributed to two factors: (1) memory constraints due to increased model size, which limited available batch sizes and reduced calibration data points, and (2) more complex memory access patterns, which made the simplifying assumptions of FLOPS-based prediction relatively less accurate. Despite some high error rates, our approach provides practical advantages by requiring only minimal calibration measurements (single workload configuration) to predict performance across diverse deployment scenarios, unlike traditional approaches [19], [20] that require exhaustive profiling of each target configuration.

While one configuration exhibits prediction error at the concerning threshold (20%), this did not impact our SLO requirements as the prediction accuracy remained sufficient for pre-deployment performance estimation and deployment configuration decisions. The significant reduction in measurement overhead—requiring only single-configuration calibration instead of exhaustive profiling across all deployment scenarios—justifies the trade-off in prediction accuracy for practical deployment planning.

C. Best Cost-Efficient Model & Architecture

Fig. 2 presents cost-efficiency heatmaps for the OPT-1.3B and OPT-2.7B models across four workload scenarios (Medium/Large workloads for each model), three GPU configurations—homogeneous (H–H), heterogeneous (H–L, ratio 0.5), and heterogeneous (H–L, ratio 0.1)—and three inference strategies: DP, AO, and MP. Each cell indicates the token-per-dollar value for a given combination, and the blue border highlights the most cost-efficient parallelization strategy for each GPU configuration.

Cost-efficiency is defined as the number of tokens served per dollar. Our analysis reveals that even within the same GPU configuration, the choice of parallelization strategy can result in significant differences in cost-efficiency.

TABLE V
OPTIMAL INFERENCE CONFIGURATION: BATCH SIZE AND LATENCY
ACROSS REPRESENTATIVE SCENARIOS (OPT-1.3B MEDIUM, OPT-2.7B
LARGE)

Scenario	GPU	Type	BatchSize	Latency (s)	Cost efficiency (tokens/$)
OPT-1.3B Medium	H-H (1:1)	**DP**	**32**	**1.95**	**315K**
		AO	64	7.54	163K
		MP	32	2.15	285K
	H-L (1:0.5)	**DP**	**32**	**3.99**	**274K**
		AO	64	7.74	212K
		MP	32	3.37	243K
	H-L (1:0.1)	DP	32	10.64	105K
		AO	**64**	**9.33**	**239K**
		MP	32	12.60	89K
OPT-2.7B Large	H-H (1:1)	DP	4	3.30	46K
		AO	8	6.60	46K
		MP	**8**	**4.04**	**76K**
	H-L (1:0.5)	DP	4	3.80	54K
		AO	8	6.77	60K
		MP	**8**	**5.33**	**77K**
	H-L (1:0.1)	DP	4	7.49	37K
		AO	**8**	**7.67**	**73K**
		MP	8	15.06	37K

The SLO is set to 9.6 seconds for the Medium workload (150 ms/token × 64 tokens) and 19.2 seconds for the Large workload (150 ms/token × 128 tokens).

For instance, in the H–H configuration with the OPT-1.3B Medium workload, DP (315 K tokens/$) achieves approximately $1.93\times$ higher efficiency than AO (163 K tokens/$). In contrast, under the H–L (1:0.1) configuration, AO (239 K tokens/$) outperforms DP (105 K tokens/$) by about $2.28\times$. A similar trend is observed for the OPT-2.7B model. In the H–H configuration with a Medium workload, MP (162 K tokens/$) shows roughly $2.03\times$ better efficiency than AO (80 K tokens/$). In the H–L (1:0.1) configuration with a Large workload, AO (73 K tokens/$) achieves about $1.97\times$ higher efficiency compared to both DP and MP (37 K tokens/$).

These findings suggest that, regardless of model size, selecting an appropriate parallelization strategy based on GPU performance ratios can significantly affect cost-efficiency. However, *this analysis overlooks the absolute cost.* From the perspective of SLO-aware cost-efficiency, it is essential to select a GPU combination and model that are not only efficient in terms of tokens per dollar but also cost-effective in absolute terms.

SLO-Aware Cost-Efficiency Strategy Selection: Table V shows FLEXLLM selecting the lowest-cost parallelization strategy for each model-workload scenario under 150ms/token SLO constraint (highlighted in yellow).

For the smaller model (OPT-1.3B), the AO strategy on the H–L (1:0.1) configuration is identified as the most cost-efficient option that satisfies the SLO constraint (latency = 9.33 seconds). Similarly, for the larger model (OPT-2.7B), the AO strategy on the H–L (1:0.1) configuration is also shown to be the cheapest strategy that meets the SLO requirement (latency = 7.67 seconds).

VII. CONCLUSION

In this study, we propose FLEXLLM, a mathematical model-based solver that identifies cost-optimal parallelization strategies for heterogeneous GPU environments under SLO constraints. Through LCF-enhanced performance modeling with minimal profiling overhead, we demonstrate that strategic architecture selection is as critical as hardware investment for economical LLM deployment.

REFERENCES

[1] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models." arXiv preprint arXiv:2303.18223, 2023. Available: https://arxiv.org/abs/2303.18223.

[2] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, and Y. Wu, "GPipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

[3] S. Chen, Y. Lin, M. Zhang, and Y. Wu, "Efficient and economic large language model inference with attention offloading." arXiv preprint arXiv:2405.01814, 2024. Available: https://arxiv.org/abs/2405.01814.

[4] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "PyTorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, pp. 3005–3018, Aug. 2020.

[5] S. M. Khan and A. Mann, "Ai chips: What they are and why they matter," tech. rep., Center for Security and Emerging Technology, Apr. 2020.

[6] H. Li, Y. Liu, Y. Cheng, S. Ray, K. Du, and J. Jiang, "Eloquent: A more robust transmission scheme for llm token streaming," in *Proc. 2024 SIGCOMM Workshop on Networks for AI Computing*, pp. 34–40, 2024.

[7] OpenAI, "Openai api pricing." [Online]. Available: https://openai.com/api/pricing, 2024. [Accessed: May 17, 2025].

[8] Anthropic, "Message batches api: Run jobs up to 50% cheaper." [Online]. Available: https://www.anthropic.com/news/message-batches-api, 2024. [Accessed: Dec. 30, 2024].

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017.

[10] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *Proc. of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 117–134, 2024.

[11] Z. Cai, X. Zhang, Z. Tan, and Z. Wei, "NQKV: A KV cache quantization scheme based on normal distribution characteristics." arXiv preprint arXiv:2505.16210, 2025. Available: https://arxiv.org/abs/2505.16210.

[12] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," in *Proc. of the 5th Conference on Machine Learning and Systems (MLSys)*, pp. 606–624, 2023.

[13] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training multi-billion parameter language models using model parallelism." arXiv preprint arXiv:1909.08053, 2019. Available: https://arxiv.org/abs/1909.08053.

[14] DeepSpeed Team, "Deepspeed pipeline parallelism tutorial." https://www.deepspeed.ai/tutorials/pipeline/, 2025. Accessed: May 22, 2025.

[15] Y. Jiang, F. Fu, X. Yao, G. He, X. Miao, A. Klimovic, B. Cui, B. Yuan, and E. Yoneki, "Demystifying cost-efficiency in llm serving over heterogeneous gpus." arXiv preprint arXiv:2502.00722, 2025. Available: https://arxiv.org/abs/2502.00722.

[16] Y. Mei, Y. Zhuang, X. Miao, J. Yang, Z. Jia, and R. Vinayak, "Helix: Serving large language models over heterogeneous gpus and network via max-flow," in *Proc. of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS)*, pp. 586–602, 2025.

[17] Y. Jiang, R. Yan, X. Yao, Y. Zhou, B. Chen, and B. Yuan, "Hexgen: Generative inference of large language model over heterogeneous environment." arXiv preprint arXiv:2311.11514, 2023. Available: https://arxiv.org/abs/2311.11514.

[18] S. Jaiswal, K. Jain, Y. Simmhan, A. Parayil, A. Mallick, R. Wang, R. S. Amant, C. Bansal, V. Rühle, A. Kulkarni, S. Kofsky, and S. Rajmohan, "Serving models, fast and slow: Optimizing heterogeneous llm inferencing workloads at scale." arXiv preprint arXiv:2502.14617, 2025. Available: https://arxiv.org/abs/2502.14617.

[19] M. Sinha, L. Vincent, M. Sand, and S. Banerjee, "FNOPerf: A robust empirical model for predicting llm performance," in *Proc. of the 31st IEEE International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, pp. 221–222, 2024.

[20] S. Imai, R. Nakazawa, M. Amaral, S. Choochotkaew, and T. Chiba, "Predicting llm inference latency: A roofline-driven ml method," in *Proc. of the Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2024.

[21] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "Flexgen: High-throughput generative inference of large language models with a single gpu," in *Proc. of the International Conference on Machine Learning (ICML)*, pp. 31094–31116, 2023.

[22] K. Rayner, E. R. Schotter, M. E. J. Masson, M. C. Potter, and R. Treiman, "So much to read, so little time: how do we read, and can speed reading help?," *Psychological Science in the Public Interest*, vol. 17, no. 1, pp. 4–34, 2016.