LLMShare: Optimizing LLM Inference Serving with Hardware Architecture Exploration

Hongduo Liu[1], Chen Bai[2], Peng Xu[1], Lihao Yin[3], Xianzhi Yu[3],
Hui-Ling Zhen[3], Mingxuan Yuan[3], Tsung-Yi Ho[1], Bei Yu[1]
[1]CUHK [2]HKUST [3]Huawei Technologies

***Abstract*—Large Language Models (LLMs) have revolutionized language tasks but pose significant deployment challenges due to their substantial computational demands during inference. The hardware configurations of existing LLM serving systems do not optimize for the different computational and bandwidth needs of the prefill and decoding phases in LLM inference, leading to inefficient resource use and increased costs. In this paper, we systematically investigate promising hardware configurations for LLM inference serving. We develop a simulator that models the performance and cost across different hardware solutions and introduce a customized design space exploration framework to identify optimal setups efficiently. By aligning hardware capabilities with the specific demands of the prefill and decoding phases, we achieve 13% cost savings and over 4× throughput improvements compared to conventional serving system setups.**

I. Introduction

Large Language Models (LLMs), such as GPTs [1], [2] and Llamas [3], [4], have achieved unprecedented performance in language understanding, generation, and complex reasoning tasks. These models have become integral to numerous applications, including chatbots [5], code generation [6], [7], and intelligent assistants [8]. As the scale of these models continues to grow, the scaling law has emerged: larger models can deliver better performance across a broad spectrum of tasks [9], [10].

However, deploying these large-scale models for inference-serving systems in real-world applications poses significant challenges. The computational demands of LLM inference require substantial GPU resources to meet stringent service-level objectives (SLOs) regarding latency and throughput. Efficiently providing LLM inference systems with performance guarantees and reducing deployment costs has become a critical problem for researchers and industry practitioners.

To address these challenges, previous research has explored various optimization strategies for LLM inference serving. Scheduling-level approaches [11]–[13] aim to maximize GPU utilization through efficient scheduling algorithms. For instance, Orca [11] introduces continuous batching at the token level, dynamically integrating new requests into the batch to improve throughput. Memory management techniques [14]–[16] focus on optimizing the use of GPU memory resources. For example, PagedAttention [14] manages the key-value (KV) cache using non-contiguous memory blocks to reduce memory fragmentation and waste.

While these approaches have made significant strides in optimizing LLM inference serving, they largely focus on software-level optimizations but merely consider the hardware configurations underlying the serving system. Recognizing this gap, a recent work Splitwise [17] proposes to split the prefill and decoding phases of LLM inference across machines with different hardware capabilities. They observe that the prefill phase is significantly more compute-intensive than the decoding phase, suggesting that using high-end GPUs like the NVIDIA H100 for both phases can lead to underutilization of hardware resources during decoding. Fig. 1 lists comparison between NVIDIA A100 cluster and H100 cluster with 8 GPUs on Llama-70B [3] without batching, where the statistics is from [17]. As we can see, although the H100 offers substantial throughput improvements (1.85×) during the prefill phase due to its increased computational power, the performance gains during the decoding phase are relatively marginal (1.43×). The discrepancy arises because the H100 exhibits limited improvements in memory size and bandwidth over the A100. This observation prompts a critical question: If we can adjust the computation capacity or memory throughput of the H100, can we achieve a better performance-cost tradeoff?

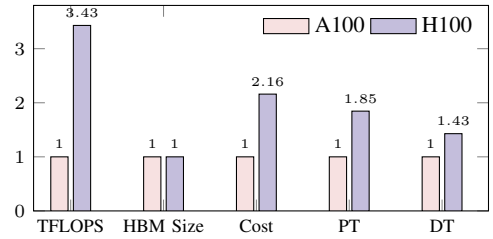

Fig. 1 Comparison of NVIDIA A100 and H100 cluster with 8 GPUs on Llama-70b without batching. 'PT' denotes prefill phase throughput, and 'DT' denotes decoding phase throughput.

Existing hardware solutions with fixed configurations, such as A100 and H100 clusters, may not perfectly match the specific computational and memory requirements of the different phases of LLM inference. This mismatch can lead to suboptimal hardware utilization and increased costs. Therefore, we are motivated to explore whether customizing hardware configurations to better align with the distinct computation and memory demands of the prefill and decoding phases can optimize both performance and cost.

In this work, we introduce **LLMShare**, a framework that optimizes large language model (LLM) inference serving through hardware architecture exploration. Our approach involves developing a simulator that models the performance and cost of different hardware configurations within an LLM serving system. Given the vast design space of LLM serving systems and the time-consuming nature of performance evaluations, we also introduce a customized design space exploration algorithm to efficiently identify optimal hardware configurations. By aligning hardware capabilities with the specific requirements of each inference phase, we achieve a 13% cost reduction and a 4× throughput improvement compared to traditional GPU setups.

The main contributions of this paper are as follows:

- We develop a simulator that models the behavior of an LLM serving system, enabling the assessment of performance and cost across various hardware configurations.
- We propose an effective design space exploration framework to identify Pareto-optimal hardware configurations within a large design space.
- We demonstrate that customized hardware configurations can achieve significant cost savings and throughput improvements over conventional GPU configurations in LLM serving.

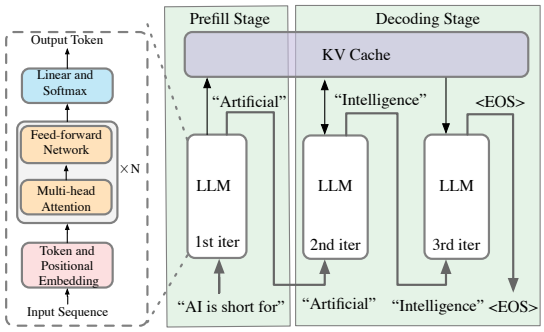

Fig. 2 The inference process of LLMs.

II. Preliminaries

A. Overview of LLM inference

Fig. 2 illustrates the inference process of decoder-only LLMs. These mainstream LLMs are predominantly constructed from multiple Transformer blocks [18], each comprising a self-attention-based multi-head attention (MHA) mechanism followed by a feed-forward network (FFN).

In the Transformer architecture, the initial step involves applying three different weight matrices to encoded embeddings of the input text sequence, which computes the queries $\boldsymbol{Q}$, keys $\boldsymbol{K}$, and values $\boldsymbol{V}$ matrices for each token. The self-attention mechanism then utilizes these matrices to calculate attention scores, allowing each token to focus on different parts of the sequence, which is given by

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}\right)\boldsymbol{V}, \tag{1}$$

where d is the dimension of the token embeddings. By employing multiple attention heads, the model can capture different types of relationships in parallel. The outputs from these multiple attention heads are concatenated and passed through a feed-forward network, which refines the information and enables the model to learn complex patterns and dependencies.

The inference process of LLMs can be systematically divided into two primary stages: the prefill phase and the decoding phase. During the prefill phase, the model processes the entire input prompt, which establishes the initial context for the LLM and generates the first token. As depicted in Fig. 2, when the LLM processes the input sequence "AI is short for", it computes the internal representations necessary to generate the next token, "Artificial". A crucial optimization in this process is the caching of the key-value (KV) pairs generated by each transformer block for each token. By storing the KV cache in memory, the model enhances computational efficiency by avoiding recomputation of these representations during subsequent decoding steps.

Following the prefill phase, the decoding phase begins, during which the model generates output tokens one at a time in an autoregressive manner. In each decoding step, the LLM predicts the next token based on both the original prompt and the sequence of tokens generated thus far. Simultaneously, it generates new KV pairs for use in subsequent iterations. For example, at the beginning of the decoding phase, the model uses the cached KV pairs to predict the next token "Intelligence". This autoregressive process ends until the LLM generates an end-of-sequence token <EOS>, signaling the completion of the generated output. By utilizing the cached KV pairs and processing tokens autoregressively, the model efficiently generates sequences without redundant computations, which is essential for handling long prompts and making real-time interactions feasible.

Parameter	Notation	Value Range	#
Server Count	sc	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	10
Device Count	dc	4, 8, 12, 16	4
Link Count Per Device	lc	6, 12, 18, 24	4
Main Memory (GB)	mm	40, 64, 80, 96, 112, 128	6
Global Buffer (MB)	gb	20, 30, 40, 50, 60, 70, 80, 90, 100, 110	10
Core Count	cc	72, 96, 108, 132, 156, 180	6
Local Buffer (KB)	lb	64, 128, 192, 256, 320, 384, 448, 512	8
Lane Count	lc	1, 2, 4, 8	4
Array Height	ah	16, 32, 64, 128	4
Vector Width	vw	16, 32, 64, 128	4

TABLE I Design space of the prefill and decoding pools. Each pool has a design space size of approximately 3×10^7. Consequently, the entire design space of an LLM serving system is nearly 9×10^{14}.

B. LLM Serving Framework

Fig. 3 illustrates the framework of a general LLM serving system. The system includes a prefill pool and a decoding pool, each consisting of multiple servers for parallel processing. The prefill pool manages the prefill phase of a request, while the decoding pool handles the completion of the decoding phase, with KV cache transfer bridging the two stages. Separating prefill and decoding servers can optimize performance [17], [19], [20], as these stages have different computational and memory communication characteristics. For instance, the prefill stage is computation-intensive and requires limited batch sizes to maintain performance, whereas the decoding phase can handle larger batch sizes, enabling higher throughput [17].

The serving system also includes a request scheduler that assigns tasks to the prefill and decoding pools. Additionally, the scheduler controls a router that dynamically reallocates server instances. If there are pending tasks in the prefill pool and no tasks in the decoding pool, instances can be moved from the decoding pool to the prefill pool. Conversely, if the prefill pool has available servers and there are blocked tasks in the decoding pool, instances can be transferred from the prefill pool to the decoding pool. Within each pool, additional scheduling handles the batching and execution of prefill or decoding tasks.

Each server (e.g., an NVIDIA DGX node) in the prefill and decoding pool comprises multiple devices connected via an inter-device interconnect (e.g., NVLink). Each device (e.g., a GPU) contains multiple cores, a shared global buffer, and off-chip main memory. The global buffer (e.g., L2 cache in NVIDIA GPUs) connects to both the main memory and the cores. Each core (e.g., a Streaming Multiprocessor in NVIDIA GPUs) includes multiple lanes that share a local buffer (e.g., L1 cache in NVIDIA GPUs). The local buffer connects to the global buffer through an on-chip interconnect. Each lane includes its vector unit, systolic or multiply-accumulation array, registers, and control logic. Here we follow the idea of LLMCompass [21] to use aforementioned general device template to describe mainstream accelerators used for LLM inference, such as NVIDIA H100 [22], AMD MI210 [23], and Google TPUv3 [24]. This description is universal because these accelerators share a similar architectural hierarchy.

This hardware template enables various LLM serving system configurations. As shown in TABLE I, we have a total design space of nearly 9×10^{14} configurations for the complete serving system. Here we only consider the server count range from 1 to 10, it can be scaled proportionally for large serving systems.

C. Problem Formulation

Definition 1 (LLM Serving System Design Space)**.** *An LLM serving system design combines a prefill pool and a decoding pool, each with design spaces detailed in TABLE I. We denote a system configuration as $\boldsymbol{x} \in \mathcal{X}$, where $\mathcal{X}$ represents the complete design space.*

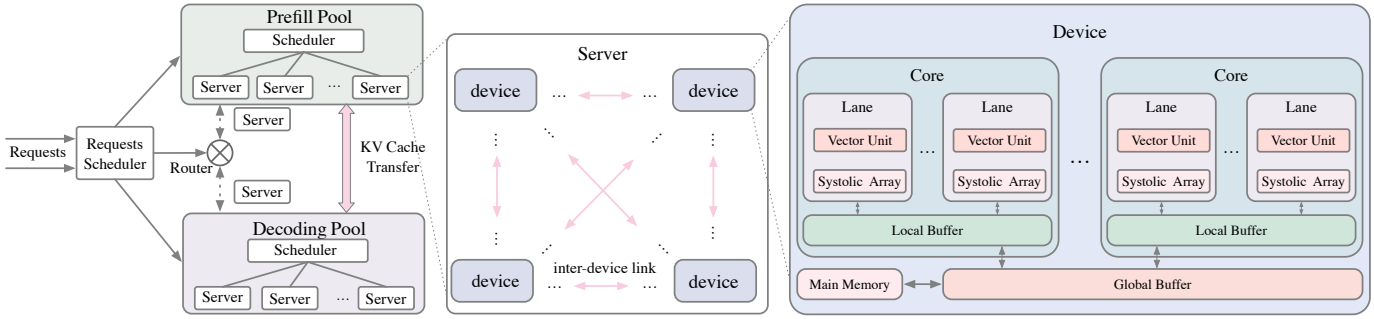

Fig. 3 Overview of a LLM serving system.

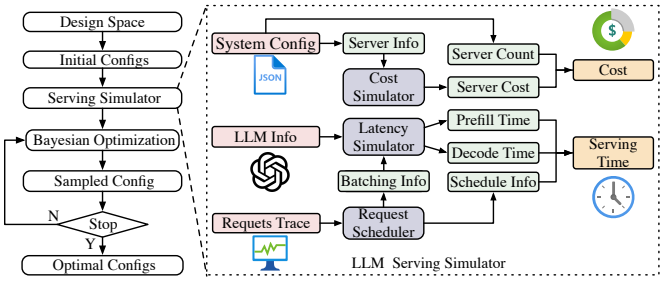

Fig. 4 The overview of LLMShare.

Definition 2 (Pareto Optimality)**.** *Given a stream of requests and an LLM, each configuration $\boldsymbol{x}$ has associated objectives: serving time $f_t(\boldsymbol{x})$ and cost $f_c(\boldsymbol{x})$. A design $\boldsymbol{x}^* \in \mathcal{X}$ is Pareto optimal if there is no other design $\boldsymbol{x} \in \mathcal{X}$ such that:*

$$f_t(\boldsymbol{x}) \leq f_t(\boldsymbol{x}^*) \quad \textit{and} \quad f_c(\boldsymbol{x}) \leq f_c(\boldsymbol{x}^*),$$

and at least one of the inequalities is strict: $f_t(\boldsymbol{x}) < f_t(\boldsymbol{x}^*)$ or $f_c(\boldsymbol{x}) < f_c(\boldsymbol{x}^*)$.

The design of LLM serving systems involves optimizing both serving time and cost. These objectives are inherently conflicting as reducing serving time typically requires additional servers, thereby increasing costs. Consequently, we focus on identifying Pareto-optimal designs.

Problem 1 (LLM Serving System Design Space Exploration)**.** *Given a design space $\mathcal{X}$, the goal of design space exploration is to identify a subset $\mathcal{X}^* \subset \mathcal{X}$ containing all Pareto-optimal configurations.*

III. LLMShare

A. Overview of LLMShare

Fig. 4 illustrates the overall workflow of LLMShare, which comprises an LLM serving simulator and a design space exploration framework. The simulator outputs the cost and serving time for a given LLM serving system configuration, providing essential feedback to the design space exploration process.

The simulator takes three main inputs: the serving system configuration, LLM information, and the request trace. The system configuration includes data on the number of servers in the prefill and decoding pools, as well as their specific configurations, as detailed in TABLE I. The LLM information covers attributes like the number of layers, the attention head count of multi-head self-attention, and hidden dimension size. The request trace details the arrival time, input token size, and output token size for each request. The cost simulator calculates the cost of a server based on its configuration, including the die costs of chips and main memory costs. By aggregating the costs of the servers in the prefill and decoding pools, the cost simulator determines the total cost of the serving system. Using the LLM information and request trace, the request scheduler decides on request batching and assigns tasks to the prefill and decoding pools. The latency simulator then calculates the prefill and decode times for each request. By considering additional scheduler details like KV cache transfer time and request pending time in the batching info reported by the request scheduler, the simulator ultimately determines the system's overall serving time. We use LLMCompass [21] for cost and latency simulation, which achieves less than a 5% error rate compared to real-world hardware. For request scheduling, we use the simulator proposed in Splitwise [17].

We adopt Bayesian optimization for design space exploration. It begins with a predefined design space, where Memory-Centric Initialization (Section III-B) is employed to generate initial configurations. Specifically, the memory size of the serving system guides the sampling of these initial designs. These configurations are then evaluated using our LLM serving simulator to obtain serving time and cost metrics based on various system configurations. To identify the optimal configurations, a deep tree kernel-based (Section III-C) Bayesian optimization algorithm is utilized to find configurations that optimize a specified acquisition function. Subsequently, the selected designs are re-evaluated through the simulator to obtain updated serving time and cost data, which are used to refine the surrogate model. Ultimately, LLMshare outputs a set of Pareto optimal designs based on the explored configurations, including initial configurations and sampled configurations during Bayesian optimization.

B. Memory-Centric Initialization (MCI)

Initial sampling involves selecting a small subset of configurations, denoted as $\mathcal{D}_x \subset \mathcal{X}$, to evaluate before commencing the iterative Bayesian optimization process. These initial points lay the groundwork for constructing the surrogate model. Evaluating each LLM serving system configuration requires processing a request trace comprising thousands of requests. Even with request batching, the simulation time remains substantial due to the frequent invocation of a computationally intensive latency simulator. Consequently, only a limited number of points are feasible for initialization.

For effective optimization, it is imperative that the initial designs are diverse and representative to ensure comprehensive coverage of the entire design space. Traditional approaches have employed random sampling [25] or orthogonal designs [26] due to their simplicity. More recently, transductive experimental design (TED) [27] has gained popularity for selecting the most representative and challenging-to-predict designs as initial points in high-level synthesis [28] and CPU microarchitecture design space exploration [29].

However, these methods do not account for the unique characteristics of LLM serving systems, leading to suboptimal initial sets.

To overcome the limitations of existing initialization methods in the context of LLM serving systems, we propose a memory-centric initialization approach. This approach is motivated by the critical role that memory configuration plays in both serving time and cost. Firstly, modern accelerators designed for LLM inference often employ high-bandwidth memory (HBM) to facilitate rapid data communication. While HBM offers substantial performance benefits, it also significantly increases the total cost of servers due to its high expense. Secondly, the memory capacity directly impacts the number of requests that can be batched and processed simultaneously. Larger memory allows for higher batching capability, which can dramatically increase throughput. Therefore, our memory-centric initialization method selects initial configurations that are more representative of the performance and cost trade-offs inherent to the LLM serving system.

Algorithm 1 Memory-Centric Initialization

Input: • $\mathcal{U}$: unsampled design space with n configurations;
• t: total number of initial configurations to select;
• u: number of groups used during sampling.
Output: $\mathcal{D}_x$ with $|\mathcal{D}_x| = t$ ▷ Selected initial designs

Compute the total main memory size for each design:
for $i = 1$ to n **do**

$$c_i = \boldsymbol{x}^p_{i,\text{sc}} \cdot \boldsymbol{x}^p_{i,\text{dc}} \cdot \boldsymbol{x}^p_{i,\text{mm}} + \boldsymbol{x}^d_{i,\text{sc}} \cdot \boldsymbol{x}^d_{i,\text{dc}} \cdot \boldsymbol{x}^d_{i,\text{mm}};$$

Determine percentiles: $P = \left\{ \frac{100 \times j}{u} \,\middle|\, j = 0, 1, \ldots, u \right\}$;
Compute bin edges for the percentiles of $\{c_i\}_{i=1}^n$:

$$B = \left\{ b_j = \text{Percentile}(\{c_i\}_{i=1}^n, p_j) \,\middle|\, j = 0, 1, \ldots, u \right\};$$

Compute base sample count per group: $q \leftarrow \left\lfloor \frac{t}{u} \right\rfloor$;
Compute remainder: $r \leftarrow t \bmod u$;
Initialize $\mathcal{D}_x \leftarrow \emptyset$;
for $j = 1$ to u **do**
  $\mathcal{G}_j = \left\{ i \,\middle|\, b_{j-1} \le c_i < b_j \right\}$;
  if $j \le r$ **then**
    $s_j \leftarrow q + 1$;
  else
    $s_j \leftarrow q$;
  Select s_j samples from $\mathcal{G}_j$: $\mathcal{S}_j = \textbf{TED}(\mathcal{G}_j, s_j)$;
  $\mathcal{D}_x \leftarrow \mathcal{D}_x \cup \mathcal{S}_j$;
return $\mathcal{D}_x$

Algorithm 1 presents our memory-centric initialization method. For each configuration, $\boldsymbol{x}_i$, a composite metric c_i is computed using main memory-related features. Specifically, c_i represents the total main memory size of configuration i,

$$c_i = \boldsymbol{x}^p_{i,sc} \cdot \boldsymbol{x}^p_{i,dc} \cdot \boldsymbol{x}^p_{i,mm} + \boldsymbol{x}^d_{i,sc} \cdot \boldsymbol{x}^d_{i,dc} \cdot \boldsymbol{x}^d_{i,mm}, \tag{2}$$

where $\boldsymbol{x}^p_{i,sc}$ denotes the server count of the prefill pool of i-th configuration while $\boldsymbol{x}^d_{i,sc}$ indicates the server count of the decoding pool of i-th configuration. The abbreviation of other features can be referred to TABLE I. Once these total memory sizes are computed, the algorithm determines percentiles to divide the range of memory sizes into equal intervals. For each interval defined by these percentiles, the algorithm identifies all designs whose total memory sizes fall within that interval. It then samples configurations using a basic sampling method like TED [27] from each interval. Finally, t samples with representative memory sizes can be obtained.

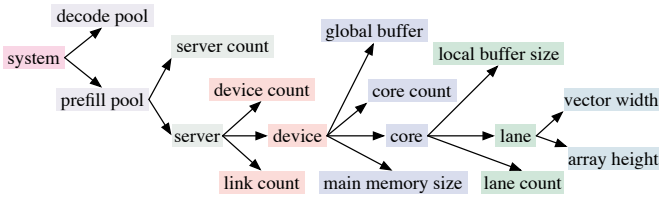

Fig. 5 Tree structure of the design space, where each leaf node represents a design parameter. Design parameters at the same hierarchical level are marked with the same color. The subtree for the decode pool is omitted as it mirrors the prefill pool's subtree.

C. Deep Tree Kernel (DKL)

After sampling the initial serving system configurations $\mathcal{D}_x$, we can utilize the simulator as shown in Fig. 4 to get serving time and cost, denoted as $\mathcal{D}_y$. Due to the time-consuming nature of metric evaluation, we aim to construct a surrogate model $\mathcal{M}$ that describes the relationship between the design space $\mathcal{X}$ and the target space $\mathcal{Y}$ based on the currently selected samples $(\mathcal{D}_x, \mathcal{D}_y)$. This surrogate model guides the selection of the next design by predicting function values at unobserved points and estimating the associated uncertainties. It is updated as the labels of newly selected samples are obtained. In this paper, we use a Gaussian process (GP) [30] as our surrogate model due to its flexibility and modeling for both predictions and uncertainty estimates, which has wide application in various design space exploration tasks [31]–[33]. A Gaussian process defines a prior over value function $f(\boldsymbol{x})$ such that any finite set of function values follows a multivariate Gaussian distribution. Formally, a GP is specified by its mean function $m(\boldsymbol{x})$ and kernel function $k(\boldsymbol{x}, \boldsymbol{x}')$:

$$f(\boldsymbol{x}) \sim \mathcal{GP}\left(m(\boldsymbol{x}), k(\boldsymbol{x}, \boldsymbol{x}')\right). \tag{3}$$

Given observed data $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, where $y_i = f(\boldsymbol{x}_i) + \epsilon_i$ and ϵ_i is Gaussian noise with variance σ_e^2, we consider the joint distribution of the function values at the observed points $\boldsymbol{f} = [f(\boldsymbol{x}_1), \ldots, f(\boldsymbol{x}_n)]^\top$ and the function value at a new point $\boldsymbol{x}_*$, denoted f_*. This joint distribution is given by:

$$\begin{bmatrix} \mathbf{f} \\ f_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mathbf{m} \\ m_* \end{bmatrix}, \begin{bmatrix} \mathbf{K_{XX}} + \sigma_e^2 \mathbf{I} & \mathbf{k_{Xx_*}} \\ \mathbf{k_{x_*X}} & k_{\mathbf{x_*x_*}} \end{bmatrix} \right), \tag{4}$$

where $\boldsymbol{m} = [m(\boldsymbol{x}_1), \ldots, m(\boldsymbol{x}_n)]^\top$, $m_* = m(\boldsymbol{x}_*)$, $\mathbf{K_{XX}}$ is the covariance matrix between the observed inputs with entries $(\mathbf{K_{XX}})_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, $\boldsymbol{k}_{\mathbf{Xx}_*} = [k(\boldsymbol{x}_1, \boldsymbol{x}_*), \ldots, k(\boldsymbol{x}_n, \boldsymbol{x}_*)]^\top$, $k_{\mathbf{x_*x_*}} = k(\boldsymbol{x}_*, \boldsymbol{x}_*)$, $\mathbf{I}$ is the identity matrix, and σ_e^2 represents the variance of the observation noise.

Choosing an appropriate kernel is crucial for the accuracy of the Gaussian process. As our design space exhibits a hierarchical tree structure as illustrated in Fig. 5, we propose a deep tree kernel tailored to capture the hierarchical structure of the hardware configuration design space for LLM serving systems. Let $T = (\mathcal{V}, \mathcal{E})$ denote a tree-structured configuration, where $\mathcal{V}$ is the set of nodes and $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is a set of directed edges that establish a parent-child relationship. Each node $v \in \mathcal{V}$ can be either a leaf node, having no children, or an internal node, with one or more children. The tree also includes a designated root node $v_r \in \mathcal{V}$, which has no parent. The root node of the tree represents the whole serving system, while each leaf node indicates a design parameter of the system. Our objective is to compute an embedding $\boldsymbol{h}_T$ that leverages the features of the

leaf nodes $\boldsymbol{x}$ and the hierarchical structure of the tree. Let $\mathcal{L} \subseteq \mathcal{V}$ denote the set of leaf nodes of T. For an internal node $v \in \mathcal{V} \setminus \mathcal{L}$, the embedding $\boldsymbol{h}_v$ is computed based on the embeddings of its child nodes. Let $\mathcal{C}(v) = \{u_1, u_2, \ldots, u_{k_v}\}$ denote the set of child nodes of node v, where k_v is the number of children of v. The embedding $\boldsymbol{h}_v$ is computed as

$$\boldsymbol{h}_v = \phi_v \left(\text{concat} \left(\boldsymbol{h}_{u_1}, \boldsymbol{h}_{u_2}, \ldots, \boldsymbol{h}_{u_{k_v}} \right); \theta_v \right), \tag{5}$$

where ϕ_v is an embedding function (e.g., an MLP) for internal node v, θ_v denotes its parameters, and concat$(\cdot)$ denotes the concatenation operation. The recursive computation proceeds in a bottom-up manner, starting from the lowest level and moving upwards through the root. At each step, we compute the embeddings of internal nodes only after computing the embeddings of all their child nodes. Finally, the embedding of the tree is given by the embedding of the root node:

$$\boldsymbol{h}_T = \boldsymbol{h}_{v_r}. \tag{6}$$

For example, the lowest-level features such as array height $\boldsymbol{x}_{\text{ah}}$ and vector width $\boldsymbol{x}_{\text{vw}}$, associated with a lane, are first transformed into lane embeddings $\boldsymbol{h}_{\text{lane}}$ using an embedding function ϕ_{lane} parameterized by θ_{lane}:

$$\boldsymbol{h}_{\text{lane}} = \phi_{\text{lane}} \left(\text{concat} \left(\boldsymbol{x}_{\text{ah}}, \boldsymbol{x}_{\text{vw}} \right); \theta_{\text{lane}} \right). \tag{7}$$

This recursive embedding culminates in the system-level embedding $\boldsymbol{h}_{\text{system}}$, which is obtained by combining the embeddings of the prefill pool $\boldsymbol{h}_{\text{prefill}}$ and decoding pool $\boldsymbol{h}_{\text{decode}}$ using the embedding function ϕ_{system} parameterized by θ_{system}:

$$\boldsymbol{h}_{\text{system}} = \phi_{\text{system}}(\boldsymbol{h}_{\text{prefill}}, \boldsymbol{h}_{\text{decode}}; \theta_{\text{system}}). \tag{8}$$

The deep tree kernel k_t between two system configurations $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ is then defined as the kernel function applied to their respective system-level embeddings:

$$k_t(\boldsymbol{x}_i, \boldsymbol{x}_j) = k \left(\boldsymbol{h}_{\text{system}}^{(i)}, \boldsymbol{h}_{\text{system}}^{(j)} \right), \tag{9}$$

where k is a traditional kernel function, such as the radial basis function. $\boldsymbol{h}_{\text{system}}^{(i)}$ and $\boldsymbol{h}_{\text{system}}^{(j)}$ denotes the system-level embedding of two configuration $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ respectively. This customized deep kernel effectively measures the similarity between different hardware configurations by leveraging the hierarchical relationships and dependencies inherent in the tree-structured design space.

D. Multi-objective Bayesian Optimization

For the design space exploration of the LLM serving system, we aim to find solutions that balance trade-offs between serving time and cost, which are two conflicting objectives. The set of these optimal trade-off solutions is known as the Pareto front. A solution is considered Pareto-optimal if improving any one objective would lead to the degradation of at least one other objective. The Pareto Hypervolume is a quantitative metric that measures the volume (or area in two dimensions) of the objective space dominated by the Pareto front, bounded by a predefined reference point. Mathematically, consider a set of n objective vectors $\mathcal{P} = \{\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(n)}\}$ in an d-dimensional objective space, where each $\mathbf{y}^{(i)} = [y_1^{(i)}, y_2^{(i)}, \ldots, y_d^{(i)}]$ represents the objective values of configuration i. Let $\mathbf{r} = [r_1, r_2, \ldots, r_d]$ be the reference point, which is chosen to be worse than any objective values in $\mathcal{P}$ (for minimization problems). The Pareto hypervolume

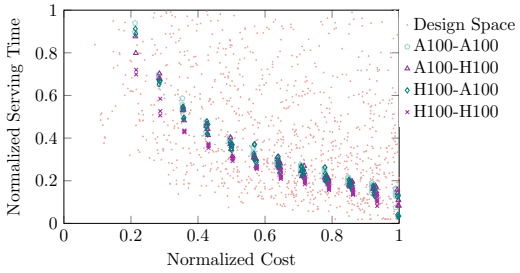

Fig. 6 The effectiveness of LLM serving system design space exploration

HV$(\mathcal{P})$ is defined as:

$$\text{HV}(\mathcal{P}) = \lambda \left(\bigcup_{i=1}^{n} [\mathbf{y}^{(i)}, \mathbf{r}] \right), \tag{10}$$

where $\lambda(\cdot)$ denotes the Lebesgue measure (length, area, volume, etc., depending on d) and $[\mathbf{y}^{(i)}, \mathbf{r}]$ represents the hyper-rectangle (in d-dimensional space) bounded by $\mathbf{y}^{(i)}$ and $\mathbf{r}$.

With the Pareto hypervolume defined, we can now discuss how it guides the selection of new design points in our multi-objective Bayesian optimization framework. We employ the Expected Hypervolume Improvement (EHVI) [34], [35] as our acquisition function to strategically explore the design space. The EIPV quantifies the expected increase in the hypervolume that would result from sampling a new design point, integrating over the uncertainty in the predictions provided by the Gaussian process (GP) surrogate model. Mathematically, for a candidate design $\mathbf{x}_*$, the expected hypervolume improvement EHV$(\mathbf{x}_*)$ is computed as:

$$\text{EHVI}(\mathbf{x}_*) = \int_{\mathbf{y}} \max\left(\text{HV}(\mathcal{P} \cup \{\mathbf{y}\}) - \text{HV}(\mathcal{P}),\, 0\right) p(\mathbf{y} \mid \mathbf{x}_*, \mathcal{D})\, d\mathbf{y}, \tag{11}$$

where $\mathcal{P}$ is the current set of Pareto-optimal objective vectors observed so far, HV$(\mathcal{P})$ is the current Pareto Hypervolume, and $p(\mathbf{y} \mid \mathbf{x}_*, \mathcal{D})$ is the predictive posterior distribution of the objective vector $\mathbf{y}$ at $\mathbf{x}_*$ given the surrogate model and observed data $\mathcal{D}$.

During the search process of Bayesian optimization, we identify the candidate design $\mathbf{x}_*$ that maximizes the EHVI:

$$\boldsymbol{x}_* = \arg\max_{\boldsymbol{x} \in \mathcal{X}} \text{EHVI}(\boldsymbol{x}). \tag{12}$$

IV. Experiments

A. Experiemtal Settings

We validate our methods using a 2-minute serving trace containing 2454 requests, with GPT3-175B [1] as the underlying LLM of the serving system. The distribution of input and output token sizes is derived from a Microsoft Azure production trace [36], reflecting real-world LLM serving demands.

As shown in TABLE I, the complete design space of an LLM serving system approaches 9×10^{14} configurations. Given the time-consuming nature of the simulation pipeline, exhaustively evaluating the entire design space is impractical. Therefore, we construct an offline dataset comprising 1,055 designs through random sampling from the complete design space. For design space exploration (DSE), we initialize with 10 sampled designs and perform 20 exploration steps. u is set to 5 for Algorithm 1.

We compare LLMShare to state-of-the-art DSE methods. DAC'16 [26] uses an AdaBoost-based strategy to selectively simulate informative designs, reducing simulation costs. ASPDAC'20 [37] employs an XGBoost model for flow parameter tuning. ICCAD'21 [29]

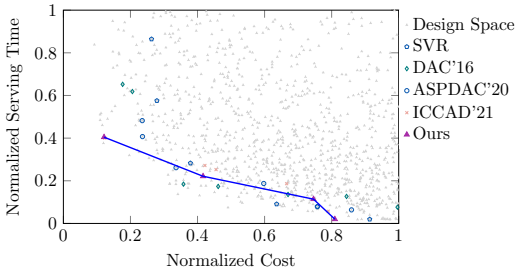

Fig. 7 Learned Pareto optimal set of LLMShare and other baseline methods.

TABLE II Comparison of different algorithms.

Algorithms	Normalized ADRS	Hypervolume (10^8)
SVR [38]	0.1811	4.9593
DAC'16 [26]	0.1718	4.9714
ASPDAC'20 [37]	0.1805	4.9513
ICCAD'21 [29]	0.2059	4.8134
LLMShare	**0.1589**	**5.0552**

integrates active learning for initial sample generation with deep-kernel learning in a Gaussian process to explore optimal RISC-V CPU designs. Additionally, a Support Vector Regression (SVR) [38] based greedy search is used as another baseline. For a fair comparison, we conduct experiments on each baseline algorithm 10 times and report the averages.

We evaluate our design space exploration method using two metrics: hypervolume, as defined in Equation (10), and Average Distance to Reference Set (ADRS). The hypervolume metric measures the size of the objective space dominated by the obtained Pareto front and bounded by a reference point. ADRS quantifies the proximity between a learned Pareto optimal set and the ground truth Pareto optimal set. The ADRS is formally defined as:

$$\text{ADRS}(\mathcal{S}, \mathcal{R}) = \frac{1}{|\mathcal{S}|} \sum_{i=1}^{|\mathcal{S}|} \min_{\boldsymbol{r} \in \mathcal{R}} d(\boldsymbol{s}_i, \boldsymbol{r}), \tag{13}$$

where $\mathcal{S}$ denotes the learned Pareto optimal set, $\mathcal{R}$ represents the groundtruth Pareto optimal set, $\boldsymbol{s}_i$ refers to each solution in $\mathcal{S}$, and $d(\boldsymbol{s}_i, \boldsymbol{r})$ measures the distance between solution $\boldsymbol{s}_i$ and true pareto optimal solution $\boldsymbol{r}$. Higher hypervolume and lower ADRS indicate better DSE algorithm performance.

B. Experiemental Results

1) Necessity of Design Space Exploration: Fig. 6 illustrates the performance comparison between randomly sampled designs from the complete design space and solutions using existing widely adopted commercial GPUs. In our notation, `A100-A100` represents configurations using NVIDIA A100 GPUs for both prefill and decoding phases, while `H100-A100` denotes configurations using NVIDIA H100 GPUs for prefill and A100 GPUs for decoding. The number of prefill and decoding servers varies from 1 to 10. Therefore, 100 total possible configurations of GPU clusters are investigated. Our analysis reveals that certain design configurations consistently outperform commercial solutions, achieving lower costs while reducing serving time. This observation highlights the benefits of exploring diverse hardware configurations to better accommodate the distinct computational and memory patterns of LLM prefill and decoding phases.

2) Performance of Proposed DSE Algorithm: Fig. 7 plots the learned Pareto optimal sets of LLMShare and other baseline algorithms. We observe that LLMShare obtains a Pareto optimal set closer to the real Pareto frontier. We also provide a quantitative comparison, as shown in TABLE II. LLMShare consistently outperforms the baseline algorithms across both ADRS and hypervolume metrics. Specifically, LLMShare achieves 12%, 7%, 11%, and 23% ADRS reduction compared with SVR [38], DAC'16 [26], ASPDAC'20 [37], and ICCAD'21 [29], respectively. The superior ADRS achieved by LLMShare demonstrates its effectiveness in providing a set of solutions that better approximate the true Pareto frontier. Additionally, LLMShare achieves the highest hypervolume, with a 5% improvement compared to ICCAD'21 [29] and a 2% improvement compared to other baselines.

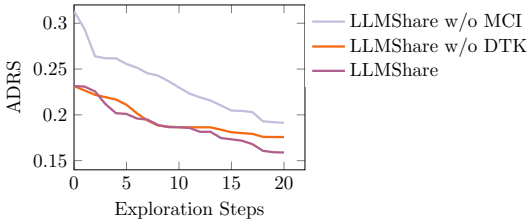

Fig. 8 Ablation study on the effectiveness of DTK and MCI.

TABLE III Normalized cost and request per second (RPS) of a Pareto optimal H100 cluster and a Pareto optimal configuration found by LLMShare. The design parameters are in the same order as TABLE I.

Hardware Config			Cost	RPS
H100-cluster	Prefill Decode	7,8,18,80,50,132,256,4,16,32 6,8,18,80,50,132,256,4,16,32	1.00	1.00
LLMShare	Prefill Decode	4,4,24,112,100,156,512,1,128,32 6,12,18,80,70,72,448,2,32,16	0.87	4.11

Fig. 8 presents an ablation study comparing different configurations of the LLMShare model over exploration steps, with ADRS on the y-axis. For `LLMShare w/o MCI`, we use the TED method implemented in ICCAD'21 [29] to obtain the initial designs instead of our Memory-Centric Initialization (MCI) algorithm. Fig. 8 reveals that the inclusion of MCI allows LLMShare to achieve a set of representative initial designs, resulting in a notably lower ADRS as exploration progresses. We also substitute the Deep Tree Kernel (DTK) with a naive deep kernel that does not consider the hierarchical design space of the LLM serving system, denoted as `LLMShare w/o DTK`. As we can see, DTK enhances the modeling of the design space, enabling more effective exploration compared to the naive deep kernel.

TABLE III lists the cost and throughput comparison between a Pareto optimal H100 cluster among the 100 possible server count combinations and a Pareto optimal configuration found by LLMShare. We can see that the learned configuration can achieve $4\times$ throughput improvement with a 13% reduction in cost.

V. Conclusion

In conclusion, this paper explores optimizing LLM inference serving through hardware design exploration. Our evaluation framework assesses the performance and cost of various configurations, showing that customized solutions tailored for prefill and decoding phases significantly enhance throughput and reduce costs compared to traditional GPUs.

Acknowledgements

The project is supported in part by Research Grants Council of Hong Kong SAR (No. RFS2425-4S02 and No. CUHK14211824), and the MIND project (MINDXZ202404).

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901.

[2] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[3] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[4] A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Yang, A. Fan *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[5] Chatgpt overview. [Online; accessed 30-October-2024]. [Online]. Available: https://openai.com/chatgpt/overview/

[6] Github copilot. [Online; accessed 30-October-2024]. [Online]. Available: https://github.com/features/copilot

[7] P. Zehua, H. Zhen, M. Yuan, Y. Huang, and B. Yu, "Betterv: Controlled verilog generation with discriminative guidance," in *Forty-first International Conference on Machine Learning*, 2024.

[8] Google assistant with bard. [Online]. Available: https://blog.google/products/assistant/google-assistant-bard-generative-ai/

[9] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[10] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling Laws for Neural Language Models," *arXiv preprint arXiv:2001.08361*, 2020.

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

[12] C. Holmes, M. Tanaka, M. Wyatt, A. A. Awan, J. Rasley, S. Rajbhandari, R. Y. Aminabadi, H. Qin, A. Bakhtiari, L. Kurilenko *et al.*, "Deepspeed-fastgen: High-throughput text generation for llms via mii and deepspeed-inference," *arXiv preprint arXiv:2401.08671*, 2024.

[13] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. S. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," *arXiv preprint arXiv:2403.02310*, 2024.

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

[15] R. Prabhu, A. Nayak, J. Mohan, R. Ramjee, and A. Panwar, "vattention: Dynamic memory management for serving llms without pagedattention," *arXiv preprint arXiv:2405.04437*, 2024.

[16] C. Hu, H. Huang, J. Hu, J. Xu, X. Chen, T. Xie, C. Wang, S. Wang, Y. Bao, N. Sun *et al.*, "Memserve: Context caching for disaggregated llm serving with elastic memory pool," *arXiv preprint arXiv:2406.17565*, 2024.

[17] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 118–132.

[18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017.

[19] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: Disaggregating prefill and decoding for goodput-optimized large language model serving," *arXiv preprint arXiv:2401.09670*, 2024.

[20] R. Qin, Z. Li, W. He, M. Zhang, Y. Wu, W. Zheng, and X. Xu, "Mooncake: Kimi's kvcache-centric architecture for llm serving," *arXiv preprint arXiv:2407.00079*, 2024.

[21] H. Zhang, A. Ning, R. B. Prabhakar, and D. Wentzlaff, "Llmcompass: Enabling efficient hardware design for large language model inference," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2024, pp. 1080–1096.

[22] Nvidia h100. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

[23] Advanced Micro Devices, Inc, "Amd cdna™ 2 architecture," AMD, Tech. Rep., 2021. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/instinct-business-docs/white-papers/amd-cdna2-white-paper.pdf

[24] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[25] B. C. Lee and D. M. Brooks, "Illustrative design space studies with microarchitectural regression models," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE, 2007, pp. 340–351.

[26] D. Li, S. Yao, Y.-H. Liu, S. Wang, and X.-H. Sun, "Efficient design space exploration via statistical sampling and adaboost learning," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[27] K. Yu, J. Bi, and V. Tresp, "Active learning via transductive experimental design," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 1081–1088.

[28] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th annual design automation conference*, 2013, pp. 1–7.

[29] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu, and M. D. Wong, "Boom-explorer: Risc-v boom microarchitecture design space exploration framework," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.

[30] D. J. MacKay *et al.*, "Introduction to gaussian processes," *NATO ASI series F computer and systems sciences*, vol. 168, pp. 133–166, 1998.

[31] Q. Sun, T. Chen, S. Liu, J. Chen, H. Yu, and B. Yu, "Correlated multi-objective multi-fidelity optimization for hls directives design," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, no. 4, pp. 1–27, 2022.

[32] H. Geng, T. Chen, Y. Ma, B. Zhu, and B. Yu, "Ptpt: Physical design tool parameter tuning via multi-objective bayesian optimization," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 42, no. 1, pp. 178–189, 2022.

[33] P. Xu, S. Zheng, Y. Ye, C. Bai, S. Xu, H. Geng, T.-Y. Ho, and B. Yu, "Ranktuner: When design tool parameter tuning meets preference bayesian optimization," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE*, 2024, pp. 1–7.

[34] S. Daulton, M. Balandat, and E. Bakshy, "Differentiable expected hypervolume improvement for parallel multi-objective bayesian optimization," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9851–9864, 2020.

[35] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. Da Fonseca, "Performance assessment of multiobjective optimizers: An analysis and review," *IEEE Transactions on evolutionary computation*, vol. 7, no. 2, pp. 117–132, 2003.

[36] Azure public dataset: Azure llm inference trace 2023. [Online]. Available: https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md

[37] Z. Xie, G.-Q. Fang, Y.-H. Huang, H. Ren, Y. Zhang, B. Khailany, S.-Y. Fang, J. Hu, Y. Chen, and E. C. Barboza, "Fist: A feature-importance sampling and tree-based method for automatic design flow parameter tuning," in *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2020, pp. 19–25.

[38] M. Awad, R. Khanna, M. Awad, and R. Khanna, "Support vector regression," *Efficient learning machines: Theories, concepts, and applications for engineers and system designers*, pp. 67–80, 2015.