LLM Inference Serving: Survey of Recent Advances and Opportunities

Baolin Li*, Yankai Jiang*, Vijay Gadepally†, Devesh Tiwari*

* Northeastern University, † MIT Lincoln Laboratory

Abstract—This survey offers a comprehensive overview of recent advancements in Large Language Model (LLM) serving systems, focusing on research since the year 2023. We specifically examine system-level enhancements that improve performance and efficiency without altering the core LLM decoding mechanisms. By selecting and reviewing high-quality papers from prestigious ML and system venues, we highlight key innovations and practical considerations for deploying and scaling LLMs in real-world production environments. This survey serves as a valuable resource for LLM practitioners seeking to stay abreast of the latest developments in this rapidly evolving field.

I. INTRODUCTION

Large language models (LLMs) have rapidly gained immense popularity since the release of ChatGPT. However, deploying and scaling these powerful AI models in production environments has presented significant challenges. The substantial computational and memory demands of LLMs often necessitate the use of high-performance GPU servers, yet even these resources can be strained by the sheer size of the models and the lengthy text sequences they process.

The growing demand for LLM-powered applications has fueled a surge of research into LLM serving systems. In this paper, we present a comprehensive survey of these systems, focusing on advancements since 2023. While previous LLM system research existed, the landscape has dramatically shifted within the last year. Nearly every major system conference now features dedicated sessions on LLMs, with a particular emphasis on serving systems due to their widespread deployment and the importance of low-latency performance for user experience.

The sheer volume of research published in such a short timeframe makes it difficult for LLM practitioners to stay abreast of developments and identify the most promising approaches for real-world deployment. This survey aims to provide a clear overview of the current state of the art, highlighting key areas of innovation and practical considerations for production environments.

In this survey, we reviewed over 100 publications and meticulously selected about 50 high-quality research papers focused exclusively on LLM serving systems, published between January 2023 and June 2024. Our selection criteria prioritized publications from prestigious machine learning (ML) and system venues (e.g., ASPLOS, MLSys, OSDI), as well as impactful arXiv submissions from established industry and academic research groups. Notably, we exclude studies that modify LLM decoding algorithms (e.g., multiple decoding head [1], lookahead decoding [2], key token selection [3]) and

solely focus on system-level enhancements that maintain the integrity of standard LLM decoding processes.

While a few prior LLM inference system surveys exist [4], [5], [6], these generally cover a broader scope and do not specifically emphasize system research. Additionally, many of the papers discussed in those surveys involve decoding algorithm modifications that can affect model accuracy. Our survey, in contrast, explicitly focuses on system-level solutions that do not alter the core LLM decoding mechanisms. Moreover, our survey encompasses a significant body of research published after the release of these earlier surveys, thus providing a more comprehensive and up-to-date overview of the field.

We have organized the recent advances in LLM serving systems into four distinct categories, each with its own set of challenges and opportunities, which we will delve into in the following sections.

KV cache and memory management. Efficient memory management is crucial to handle the dynamic growth of KV caches, which store previous key-value pairs to accelerate LLM inference. Recent research explores non-contiguous memory allocation, distributed management, and intelligent caching strategies to optimize memory utilization. Compression techniques are also being investigated to reduce the overall memory footprint, ultimately enhancing LLM performance and scalability by allowing for longer context lengths and lower memory overhead.

LLM computation optimization. Efforts to optimize LLM computation focus on request batching to maximize resource utilization. Additionally, disaggregating the inference process into prefill and decode phases enables independent optimization and hardware specialization. Model parallelism, employing various techniques, facilitates efficient execution across multiple GPUs. These strategies collectively enhance LLM execution efficiency and hardware utilization.

Cloud LLM deployment. Cloud platforms provide a scalable and cost-effective foundation for LLM inference. However, challenges remain in optimizing costs and resource utilization. Research is addressing this through techniques such as spot instance management, serverless optimizations, intelligent resource allocation, and power management. Additionally, strategies like cloud task co-location and token delivery optimization enhance user experience and overall cloud efficiency.

Emerging research fields. Emerging areas in LLM serving include retrieval-augmented generation (RAG) and mixture-of-experts (MoE) inference. RAG faces challenges related to
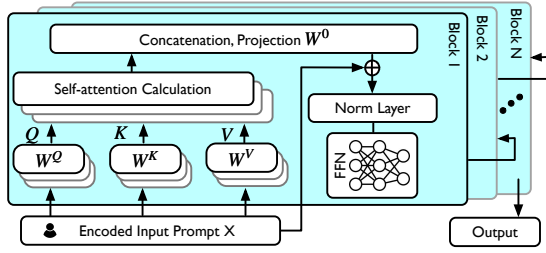
Fig. 1: Transformer-based LLM architecture including both the multi-head attention mechanism and feed-forward network.

Fig. 2: Prefill and decoding phase in the LLM inference.

the computational overhead of increased input lengths due to retrieved documents, while MoE inference grapples with efficient communication and load balancing between distributed experts. Other research efforts address ethical concerns in LLM serving, such as fairness and environmental sustainability, for which we provide a comprehensive list of relevant studies.

II. BACKGROUND

A. Overview of Transformer-based LLM Architecture

Mainstream LLMs are built on multiple transformer blocks [7]. Each identical transformer primarily consists of self-attention-based *Multi-head Attention* (MHA) operations and *Feed-Forward Networks* (FFN). Initially, the transformer applies three weight matrices (W^Q, W^K, W^V) to the input X (encoded representation of input text sequence) to compute queries Q, keys K, and values V. Then, the *Self-attention* is calculated as:

$$Q = XW^Q; K = XW^K; V = XW^V$$

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

This is the calculation of one attention head (H_i), and multiple heads are concatenated and linearly projected into the final attention result:

$$H_i = \text{Attention}(XW_i^Q, XW_i^K, XW_i^Q)$$

$$\text{Multi-Head Attention} = \text{Concat}(H_1, H_2, ..., H_h)W^O$$

MHA makes transformers focus on different parts of the sequence in different representational spaces. Next, following the MHA block, the normalized output is fed into a position-wise FFN, which consists of two linear transformations with a ReLU activation.

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

The FFN can be applied separately to each position, further refining the information captured by the MHA block. The output will have the same dimension as the input X. Fig. 1 provides a visualization of the LLM architecture.

B. Overview of LLM Inference

LLM inference generates output tokens autoregressively [8] based on the initial input sequences P, referred to as *Prompts*. This process is divided into two major phases: the prefill phase and the decoding phase. The prefill phase is essential for setti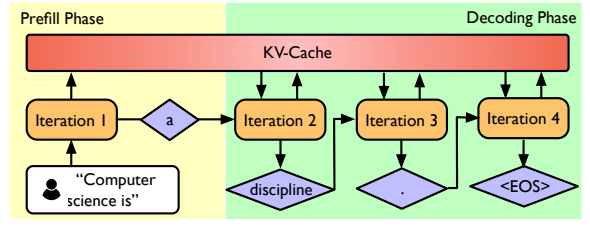ng up the model to generate text efficiently, while the decoding phase handles the generation of subsequent tokens. We visualize this process until it reaches the End-of-Sequence token (EOS) in Fig. 2.

The prefill phase starts with a tokenized and encoded representation of the prompt going through layers of the transformers. Note that the generated key-value (KV) pairs of all transformer blocks are cached during the prefill phase, referred to as KV cache [9]. It ensures that the model can generate tokens more efficiently without recomputing the KV vectors of all previous tokens. Let the input prompt $P = [p_1, p_2, ..., p_n]$, during the prefill phase, a new token is generated, denoted as P_{n+1}, and the new K and V are cached as $[(k_1, v_1), (k_2, v_2), ..., (k_n, v_n)]$.

The decoding phase is where the model generates new tokens autoregressively. The LLM predicts the next token, appends the newly generated token p_{n+1} to the original prompt P, and updates the KV cache. Note that the KV cache grows linearly with the number of tokens generated. The autoregressive LLM inference process is outlined in Algorithm 1.

Algorithm 1 Autoregressive LLM Inference

    Input P: encoded input sequence $[p_1, p_2, ..., p_n]$
    Output X: generated new sequence $[]$.
1: Forward Pass $([p_1, p_2, ..., p_n])$
2: Store the KV cache: $[(k_1, v_1), (k_2, v_2), ..., (k_n, v_n)]$
3: **for** i from 1 to M **do**
4:     Predict the next token p_{n+i} using the KV cache.
5:     Store (k_{n+i}, v_{n+i}) to the KV cache.
6:     $X \leftarrow X \cup \{p_{n+i}\}$
7:     **if** p_{n+i} is EOS token **or** len(X)>max length **then**
8:         **break**

III. MEMORY MANAGEMENT AND CACHING

In this section, we explore memory management techniques to mitigate memory footprint and access overhead during LLM inference. While model parameters remain constant and intermediate activations are relatively small, the KV cache – used to store attention information – grows substantially with the number of generated tokens. Therefore, recent research has focused on efficient KV cache management to enable larger batch sizes and longer context processing.

A. Efficient Management of KV Cache

PagedAttention [10] identifies that the KV cache dynamically grows and shrinks over time as the model generates new tokens, but the request generation lifetime and length are not known a priori. Thus, it proposes to manage the

KV cache as non-contiguous memory blocks. Compared to contiguous KV cache, non-contiguous KV cache management significantly reduces the memory waste on pre-allocation and fragmentation. Due to its efficient memory management using pages, PagedAttention has become an industry norm in LLM serving frameworks, supported by TGI [11], vLLM [10] and TensorRT-LLM [12].

Despite its success, researchers still identify its weakness as PagedAttention requires rewriting attention kernels to accommodate the non-contiguous memory blocks, its memory manager adds software complexity and redundancy, and introduces performance overhead. Recently, vAttention [13] was proposed to retain the KV cache in contiguous virtual memory. It leverages pre-existing low-level system calls for demand paging, which is a standard operating system feature to reduce the software complexity. vAttention overlaps memory allocation with computation, pre-allocates memory ahead of time, and defers memory reclamation to hide the latency of memory allocation and improve the overall performance of the system.

Besides system memory management, other efforts have addressed application-specific KV cache efficiency. Prompt Cache [14] designs specific prompt schema for users to submit their requests, so that attention states from these pre-defined modules (e.g., system prompt) can be reused across multiple prompts. AttentionStore [15] identifies that human interactions with applications such as ChatGPT are mostly multi-turn conversations. However, LLM engines would discard the KV cache when the user session becomes inactive to free up HBM space for other active sessions and re-compute the whole KV cache again when the session becomes active, leading to extra pre-filling costs. AttentionStore utilizes slower-mediums (e.g., CPU memory and disk), overlaps KV cache loading with computation, and designs intelligent pre-fetching and eviction policies.

B. Support for Long-Context Applications

Serving long-context LLM applications is particularly challenging as the size of the KV cache scales with the number of tokens. The limited memory limits LLM's ability to handle long sequences, demanding more memory-efficient solutions. Ring attention [16] is a novel distributed approach that leverages blockwise computation of attention and feedforward of long sequences across multiple devices. It efficiently overlaps KV cache communication with computation and extends the context length by the device count times. Infinite-LLM [17] is another distributed solution, it breaks down KV cache into smaller manageable units called rBlocks across GPUs/CPUs, and efficiently manages them with dynamic memory sharing and coordination. MemServe [18] unifies handling of inter-request and intra-request optimizations for LLM serving by introducing MemPool, a distributed memory pool to manage KV cache across all cluster memory and employs a global scheduler to maximize KV cache reuse.

When the context grows larger than the GPU memory limit, most systems offload the KV cache to the CPU. InfiniGen [19]

is a solution that speculates the important KV cache entries by rehearsing the attention computation of the current layer in the preceding layer and prefetches only the essential entries to the GPU, thereby reducing the data transfer overhead. LoongServe [20] introduces a new parallelism paradigm called Elastic Sequence Parallelism (ESP) to dynamically adapt to resource usage variance between requests and phases (prefilling and decoding) of a request. It reduces KV cache migration overhead and KV cache fragmentation when serving long sequences.

C. Compression of KV Cache

Due to the large memory footprint of LLM serving, some systems have resorted to compressing the KV cache. On top of memory aggregation and communication scheduling, FlexGen [21] uses fine-grained groupwise quantization to compress the weights and KV cache to 4 bits. KIVI [22] analyzes the element distribution of the LLM KV cache and applies asymmetric quantization of the Key and Value cache. KIVI quantizes the key cache per-channel (grouping elements along the channel dimension) and the value cache per-token to achieve minimum quantization error. Gear [23] achieves near-lossless high-ratio KV cache compression by quantizing the majority of entries of similar magnitudes and employs a low-rank matrix to approximate the quantization error. MiniCache [24] observes that the KV cache states exhibit high similarity between adjacent layers in the middle-to-deep portion of LLMs. Based on this insight, MiniCache leverages this high similarity to merge them into a shared representation to reduce redundancy, while also identifying and retaining distinct states that are crucial for maintaining the model's performance, preventing information loss during compression.

IV. COMPUTATION TASK SCHEDULING

Besides memory and KV cache management, the computation of LLM also presents significant system challenges. Due to the sequential dependency between tokens during the autoregressive generation, LLM can only generate one token at a time for each request. Thus, LLM inference workloads are less resource-efficient than training workloads on GPU hardware that is designed for massively parallel execution. Following this incentive, we investigate system solutions that optimize the scheduling of computation tasks during the inference process.

A. Request Batching

When a single request cannot efficiently utilize the GPU, it is intuitive to batch multiple inference requests together to boost the occupancy of GPU cores. However, as responses to different prompts can have significantly variable lengths, when batched together, the shorter responses are forced to wait for the longer ones to complete, resulting in computational waste. Response Length Perception and Sequence Scheduling [25] instructs the LLM to predict the response length before starting to generate the actual response, and batches queries with

similar predicted response lengths to reduce computational waste. A similar approach, S^3 [26], finetunes a Distillbert model for sequence length prediction. Upon mispredictions, it preempts sequences that exceed their allocated memory and retrain the predictor to learn from its mistakes.

Generation length prediction based batching is less practical due to the strong reliance on the predictor. Orca [27] proposes continuous batching at the token level rather than the request level. It continuously schedules new requests into the batch as soon as a request in the current batch completes. Continuous batching now has become an industry standard in LLM serving frameworks, incorporated into the software of TGI, vLLM, and TensorRT-LLM. Based on continuous batching, DeepSpeed-FastGen [28] proposes a dynamic SplitFuse mechanism that decomposes long prompts into smaller chunks scheduled across multiple iterations and composes short prompts together to maintain the inference running at high throughput region (bounded by GPU compute not memory bandwidth). A similar idea was explored in Sarathi-Serve [29], which splits prefill requests into smaller chunks and schedules them alongside ongoing decode requests without causing stalls (stall-free batching). This allows new requests to join a running batch without pausing ongoing decodes, leading to minimal pipeline bubbles.

B. Disaggregated Inference

LLM inference goes through a prefill stage to process the prompt, populate the KV cache, and start the decoding stage to generate tokens (Sec. II). Existing LLM serving systems colocate the two phases and batch the computation of prefill and decoding across all users and requests. However, these two phases display distinct characteristics and can interfere with each other when requests at the prefill stage are batched with requests at the decoding stage. TetriInfer [30] separates prefill and decode instances, allowing each phase to run independently and preventing interference between batch-like prefill jobs and latency-critical decode tasks. It employs a two-level scheduling algorithm that incorporates predicted resource usage to avoid scheduling hotspots during the decode phase, ensuring efficient resource allocation and minimizing contention.

Splitwise [31] extensively characterizes the differences in the execution and utilization patterns of the prefill and decoding stage on different generations of GPUs (heterogeneous hardware). Splitwise proposes to split these two phases into separate machines, allowing for specialized hardware for each phase to achieve better utilization, reduce hardware ownership costs, and save energy. DistServe [32] designs a placement algorithm to schedule the prefill and decoding stage computation tasks. In clusters with high-speed cross-node networks, DistServe optimizes parallelism configurations for prefill and decoding instances independently to achieve the best per-GPU goodput; In clusters with limited cross-node bandwidth, it ensures that prefill and decoding instances of the same stage are co-located within a single node and optimizes parallelism configurations within the node.

C. Model Parallelism

LLMs can have hundreds of billions of parameters, requiring model parallel execution on multiple GPUs. Pope et al. [9] develop an analytical model for inference efficiency, enabling the selection of optimal multi-dimensional partitioning techniques tailored for TPU v4 slices based on specific application needs. HeteGen [33] introduces a framework for heterogeneous parallel computing using CPUs and GPUs. It employs a heterogeneous parallel computing algorithm to distribute computation within its hybrid heterogeneous parallelism framework and enables asynchronous overlap to mitigate I/O bottlenecks between the CPU and GPU.

ExeGPT [34] can find an optimal schedule control variable of the batch size and tensor parallelism degree that maximizes inference throughput while adhering to a given latency limit. It leverages the distribution of input and output sequence lengths to allocate resources efficiently and determine the best parallelism configuration. Helix [35] is designed to partition an LLM across heterogeneous GPUs and different types of network connections. It formulates its model partition scenario as a max-flow problem of a directed, weighted graph whose nodes represent GPU instances and edges capture both GPU and network heterogeneity through their capacities in the max-flow problem.

V. LLMs in the Cloud

LLM deployments are computationally intensive and often require significant infrastructure to run effectively. Cloud platforms offer a scalable and cost-effective solution for deploying LLMs, eliminating the need for expensive hardware investments. The flexibility of cloud deployment allows organizations to easily adjust resources as needed, ensuring optimal performance and minimizing downtime. However, the significant costs associated with cloud computing resources and the challenge of ensuring their efficient utilization can be major obstacles for LLM service providers.

A. Cloud Deployment Cost

Modern clouds offer a variety of spot instances (e.g., AWS EC2 Spot Instance, Azure Spot Virtual Machines, Google Cloud Spot VMs). These instances run on spare capacity and are offered at highly discounted prices, but may be preempted at any time when other instances need the capacity. Spot-Serve [36] addresses the challenges of using these instances for LLM serving, such as how to quickly adapt to changes in available instances and how to minimize the cost of migrating instances when interruptions occur. It also introduces a stateful inference recovery mechanism for inference engines to commit their progress at the token level and efficiently resume interrupted requests.

Serverless is a recently emerged cloud computing paradigm, where inference service users can submit their model to the cloud and the cloud provider takes care of all infrastructure provision and scaling with varying inference request load, and saves unused hardware costs for customers. A major challenge in serverless is mitigating cold start, where a service

instance would be shut down after not being accessed for some time, and once a new request arrives, it would experience a latency spike associated with re-initializing the service instance. ServerlessLLM [37] addresses these latency issues by utilizing the underutilized storage and memory resources available on GPU servers. It introduces a new checkpoint format and loading system to speed up LLM model loading, a live migration mechanism to avoid interrupting ongoing inferences, and a locality-aware server allocation strategy to minimize LLM inference cold start latency.

Cloud providers often offer a wide range of heterogeneous instance selections labeled at different prices. Mélange [38] is a cloud resource allocation framework that considers three key LLM service characteristics: request size, request rate, and service-level objective. It automatically navigates through the GPU option space to determine the most cost-efficient heterogeneous GPU allocation for a given LLM service. With the resources allocated and model hosted on the GPUs, Llumnix [39] is a dynamic scheduling system for LLM serving that addresses the challenges of heterogeneous and unpredictable requests by rescheduling them across multiple model instances at runtime – similar to how OS context switches across cores. Llumnix introduces an efficient live migration mechanism for requests and their in-memory states, minimizing downtime during rescheduling, and employs a dynamic scheduling policy that unifies various rescheduling scenarios, such as load balancing, de-fragmentation, prioritization, and auto-scaling. This efficiency has resulted in significant cost savings while achieving similar tail latency.

B. Cloud Efficiency

A key bottleneck resource in cloud datacenters is power, which LLMs are quickly saturating due to their growing computation demand. POLCA [40] characterizes the power consumption patterns of LLMs in the cloud and finds that while training LLMs demands a lot of power and can strain the data center's power infrastructure, inference tasks offer more flexibility for power management due to their less predictable power demands. POLCA devises a framework to manage power in LLM inference clusters by dynamically applying techniques such as GPU frequency locking and power capping. PerLLM [41] takes the LLM inference to an edge-cloud collaboration scenario, where it leverages the strengths of edge computing (low latency, reduced energy costs) and cloud computing (high processing power) to handle LLM inference tasks efficiently. PerLLM employs a Constraint Satisfaction Upper Confidence Bound (CS-UCB) algorithm to optimize service scheduling and resource allocation while adhering to constraints like processing time, bandwidth, and computing power – achieving energy LLM efficiency.

Workloads often get co-located in the cloud environment. FlexLLM [42] is a system designed to efficiently service LLM inference and parameter-efficient fine-tuning (PEFT) requests in the same iteration. LLM inference, which involves generating text token by token, is primarily limited by memory bandwidth due to the need to access all model parameters for each token generation. In contrast, PEFT, which processes all tokens of a request simultaneously, is mainly constrained by compute resources, such as the tensor cores on GPUs. FlexLLM introduces a token-level fine-tuning mechanism that breaks down the fine-tuning process into smaller, more manageable token-level computations to minimize memory usage and inference latency, making co-serving feasible.

As LLM inference follows token-by-token generation, users also read the response word-by-word. Andes [43] defines a user experience metric of Quality of Experience (QoE) for text streaming services. It is formulated by comparing the actual token delivery timeline (TDT) of a request with its expected TDT. The expected TDT is determined by the expected time to first token (TTFT) and the expected token delivery speed (TDS), which can vary depending on factors like the user's typical reading speed. The intuition is generating text too fast (than user reading speed) does not yield QoE benefits, wasting cloud resources. Andes addresses this by strategically allocating GPU resources among multiple requests to optimize QoE. It employs a dynamic priority-based preemptive scheduler that operates at the token level, prioritizing urgent requests and preempting those that have been sufficiently served. Andes improves average QoE and can handle higher request rates while maintaining similar token generation throughput.

VI. EMERGING RESEARCH FIELDS

A. Retrieval Augmented Generation

Retrieval-Augmented Generation (RAG) [44] is a technique that enhances LLMs by incorporating external information sources. It addresses the limitations of LLMs in retaining factual knowledge and their tendency to generate inaccurate or fabricated information (hallucinations) [45]. RAG operates in two stages: retrieval and generation. During retrieval, the system identifies the most relevant contexts from an external knowledge base or corpus based on the given query. Once the relevant contexts are retrieved, they are integrated into the LLM's generation process in different processes including concatenation (where the retrieved contexts are simply appended to the query) and cross-attention (where the LLM attends to the retrieved contexts during generation).

Sparse RAG [46] observes that RAG can be computationally expensive due to the increased input length from retrieved documents. It first encodes retrieved documents in parallel to eliminate latency caused by long-range attention, then selectively decodes the output by attending only to highly relevant caches chosen via prompting the LLM with special control tokens. RAGCache [47] caches intermediate states of external knowledge with a knowledge tree to organize and store intermediate states. The cached knowledge can be shared across multiple queries to reduce the redundant computation. Another knowledge caching technique is CacheBlend [48], which selectively recomputes a small portion of the KV cache based on the preceding text in the input.

B. Mixture-of-Experts Inference

The mixture of Experts (MoE) is used in LLMs to improve efficiency and performance. It divides the model into specialized sub-networks, called "experts", each focusing on a specific task. A "gating" network then directs input to the most suitable expert. In the inference process of an MoE transformer, the input is first passed through a gating network. This network determines which expert, or a combination of experts, is best suited to process the specific input. MoE's sparsely activated subset of experts avoids the large computational need to process the entire model for every inference.

MoE Communication. Lina [49] is a system designed to address the all-to-all communication bottleneck in distributed MoE. The all-to-all communication occurs when distributed MoE sends tokens to their selected experts for processing and then sends the results back to the original devices. During inference, Lina dynamically schedules resources based on expert popularity, balancing the transfer size and bandwidth of all-to-all communication across devices. ExFlow [50] is an optimization technique to accelerate the inference of distributed MoE. It leverages the inter-layer expert affinity, which is the correlation between expert selection across different MoE layers. By placing experts on corresponding GPUs based on their affinity, ExFlow reduces cross-GPU routing latency and improves inference throughput.

Expert offloading. SiDA-MoE [51] (Sparsity-inspired Data-Aware) leverages both main memory and GPU memory by exploiting the inherent sparsity of expert activation in MoE models. SiDA-MoE includes two parallel threads: an inference thread and a hash-building thread. The hash-building thread predicts which experts will be activated for each token at each layer, storing these predictions in a hash table. The inference thread then uses this information to dynamically load activated experts onto the GPU and offload inactive experts to main memory, maximizing GPU memory utilization. MoE-Infinity [52] takes a different approach toward expert offloading. The system leverages the observation that MoE models exhibit sparse activation and temporal locality during inference, meaning only a few experts are repeatedly activated for processing a specific sequence. MoE-Infinity traces expert activation at the sequence level, enabling it to predict which experts will be needed and prefetch them accordingly.

MoE Efficiency. Fiddler [53] is a system designed to efficiently run these models on a limited number of GPUs, even when the model's size would typically exceed the GPU's memory capacity. Fiddler strategically distributes the model's components. Non-expert layers, which are used frequently, are kept on the GPU. A subset of expert layers, chosen based on how often they're used, are also placed on the GPU. The rest remain in the CPU's memory. Huang et al. [54] introduce three optimization techniques to address the MoE inference inefficiencies. (i) Dynamic gating allows the number of tokens processed by each expert to vary, which avoids the over-provisioning of resources in static gating and reduces computational waste, communication overhead, and memory

consumption. (ii) Expert buffering leverages the observation that expert activation is often sparse and exhibits temporal locality. By caching frequently used (hot) experts in GPU memory and buffering less active experts in CPU memory, expert buffering reduces the static memory allocation on GPU. (iii) Imbalanced token assignments to experts can lead to bottlenecks and performance degradation. Expert load balancing ensures a more even distribution of workload across devices.

C. Miscellaneous Fields

Ethics and environmental sustainability. Sheng et. al [55] ensure fairness in serving LLMs by introducing a Virtual Token Counter (VTC). VTC defines LLM serving fairness based on a cost function that accounts for the number of input and output tokens processed. It achieves fairness by tracking the services received by each client and prioritizing those with the least service, while also considering the varying costs of processing input and output tokens. Sprout [56] addresses the environmental sustainability of LLMs and designs a framework to reduce the carbon footprint of LLM inference services. Sprout introduces "generation directives" to guide the autoregressive generation process, balancing the need for sustainability with the demand for high-quality generation.

End-to-end optimization. FlashDecoding++ [57] improves performance in softmax synchronization, GPU kernel, and dataflow. For example, the decoding phase performs linear GEMM operations with flat shapes where the batch size dimension involved in the multiplication is much smaller than the others. FlashDecoding++ accelerates flat GEMM with double buffering that overlaps computation and data transfer and hides the memory latency in loading input matrices. Parrot [58] is designed to optimize the performance of LLM-based applications that involve multiple LLM requests with complex workflows. Parrot performs data flow analysis and uncovers correlations across multiple LLM requests. FlashAttention-3 [59] introduces techniques like warp specialization and asynchronous block-wise operations to optimize GPU utilization.

Frugal inference. FrugalGPT [60] proposes several solutions to reduce the inference cost, such as prompt caching and LLM cascading which uses a sequence of LLMs, starting with cheaper ones and moving to more expensive ones only if necessary. SpecInfer [61] applies speculative decoding using smaller, speculative models to predict the LLM's output, reducing the computational resources. These predictions are organized into a tree structure, and their accuracy is verified in parallel against the LLM. RouteLLM [62] dynamically selects between a stronger and a weaker LLM during inference to optimize the balance between cost and response quality.

VII. Conclusion

This survey has presented a comprehensive overview of recent advancements in LLM serving systems, emphasizing the importance of system-level solutions for enhancing performance and efficiency. We have highlighted key innovations for deploying and scaling LLMs, paving the way for the future development of LLM serving systems.

REFERENCES

[1] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao, "Medusa: Simple llm inference acceleration framework with multiple decoding heads," *arXiv preprint arXiv:2401.10774*, 2024.

[2] Y. Fu, P. Bailis, I. Stoica, and H. Zhang, "Break the sequential dependency of llm inference using lookahead decoding," *arXiv preprint arXiv:2402.02057*, 2024.

[3] M. Adnan, A. Arunkumar, G. Jain, P. Nair, I. Soloveychik, and P. Kamath, "Keyformer: Kv cache reduction through key tokens selection for efficient generative inference," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 114–127, 2024.

[4] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, H. Jin, T. Chen, and Z. Jia, "Towards efficient generative large language model serving: A survey from algorithms to systems," *arXiv preprint arXiv:2312.15234*, 2023.

[5] Z. Yuan, Y. Shang, Y. Zhou, Z. Dong, C. Xue, B. Wu, Z. Li, Q. Gu, Y. J. Lee, Y. Yan *et al.*, "Llm inference unveiled: Survey and roofline model insights," *arXiv preprint arXiv:2402.16363*, 2024.

[6] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li *et al.*, "A survey on efficient inference for large language models," *arXiv preprint arXiv:2404.14294*, 2024.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[8] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[9] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 606–624, 2023.

[10] 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.

[11] "Text generation inference: A rust, python and grpc server for text generation inference." [Online]. Available: https://github.com/huggingface/text-generation-inference

[12] "Tensorrt-llm: A tensorrt toolbox for optimized large language model inference." [Online]. Available: https://github.com/NVIDIA/TensorRT-LLM

[13] 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.

[14] I. Gim, G. Chen, S.-s. Lee, N. Sarda, A. Khandelwal, and L. Zhong, "Prompt cache: Modular attention reuse for low-latency inference," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 325–338, 2024.

[15] B. Gao, Z. He, P. Sharma, Q. Kang, D. Jevdjic, J. Deng, X. Yang, Z. Yu, and P. Zuo, "Cost-Efficient large language model serving for multi-turn conversations with CachedAttention," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 111–126. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/gao-bin-cost

[16] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," *arXiv preprint arXiv:2310.01889*, 2023.

[17] B. Lin, T. Peng, C. Zhang, M. Sun, L. Li, H. Zhao, W. Xiao, Q. Xu, X. Qiu, S. Li *et al.*, "Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache," *arXiv preprint arXiv:2401.02669*, 2024.

[18] 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.

[19] W. Lee, J. Lee, J. Seo, and J. Sim, "Infinigen: Efficient generative inference of large language models with dynamic kv cache management," *arXiv preprint arXiv:2406.19707*, 2024.

[20] B. Wu, S. Liu, Y. Zhong, P. Sun, X. Liu, and X. Jin, "Loongserve: Efficiently serving long-context large language models with elastic sequence parallelism," *arXiv preprint arXiv:2404.09526*, 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 *International Conference on Machine Learning*. PMLR, 2023, pp. 31 094–31 116.

[22] Z. Liu, J. Yuan, H. Jin, S. Zhong, Z. Xu, V. Braverman, B. Chen, and X. Hu, "Kivi: A tuning-free asymmetric 2bit quantization for kv cache," *arXiv preprint arXiv:2402.02750*, 2024.

[23] H. Kang, Q. Zhang, S. Kundu, G. Jeong, Z. Liu, T. Krishna, and T. Zhao, "Gear: An efficient kv cache compression recipefor near-lossless generative inference of llm," *arXiv preprint arXiv:2403.05527*, 2024.

[24] A. Liu, J. Liu, Z. Pan, Y. He, G. Haffari, and B. Zhuang, "Minicache: Kv cache compression in depth dimension for large language models," *arXiv preprint arXiv:2405.14366*, 2024.

[25] Z. Zheng, X. Ren, F. Xue, Y. Luo, X. Jiang, and Y. You, "Response length perception and sequence scheduling: An llm-empowered llm inference pipeline," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[26] Y. Jin, C.-F. Wu, D. Brooks, and G.-Y. Wei, "S3: Increasing gpu utilization during generative inference for higher throughput," *Advances in Neural Information Processing Systems*, vol. 36, pp. 18 015–18 027, 2023.

[27] 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.

[28] 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.

[29] 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.

[30] C. Hu, H. Huang, L. Xu, X. Chen, J. Xu, S. Chen, H. Feng, C. Wang, S. Wang, Y. Bao *et al.*, "Inference without interference: Disaggregate llm inference for mixed downstream workloads," *arXiv preprint arXiv:2401.11181*, 2024.

[31] P. Patel, E. Choukse, C. Zhang, Í. Goiri, A. Shah, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," *arXiv preprint arXiv:2311.18677*, 2023.

[32] 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.

[33] Z. XUANLEI, B. Jia, H. Zhou, Z. Liu, S. Cheng, and Y. You, "Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 162–172, 2024.

[34] H. Oh, K. Kim, J. Kim, S. Kim, J. Lee, D.-s. Chang, and J. Seo, "Exegpt: Constraint-aware resource scheduling for llm inference," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 369–384.

[35] Y. Mei, Y. Zhuang, X. Miao, J. Yang, Z. Jia, and R. Vinayak, "Helix: Distributed serving of large language models via max-flow on heterogeneous gpus," *arXiv preprint arXiv:2406.01566*, 2024.

[36] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 1112–1127.

[37] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "Serverlessllm: Locality-enhanced serverless inference for large language models," *arXiv preprint arXiv:2401.14351*, 2024.

[38] T. Griggs, X. Liu, J. Yu, D. Kim, W.-L. Chiang, A. Cheung, and I. Stoica, "M\`elange: Cost efficient large language model serving by exploiting gpu heterogeneity," *arXiv preprint arXiv:2404.14527*, 2024.

[39] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," *arXiv preprint arXiv:2406.03243*, 2024.

[40] P. Patel, E. Choukse, C. Zhang, Í. Goiri, B. Warrier, N. Mahalingam, and R. Bianchini, "Characterizing power management opportunities for llms in the cloud," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 207–222.

[41] Z. Yang, Y. Yang, C. Zhao, Q. Guo, W. He, and W. Ji, "Perllm: Personalized inference scheduling with edge-cloud collaboration for diverse llm services," *arXiv preprint arXiv:2405.14636*, 2024.

[42] X. Miao, G. Oliaro, X. Cheng, M. Wu, C. Unger, and Z. Jia, "Flexllm: A system for co-serving large language model inference and parameter-efficient finetuning," *arXiv preprint arXiv:2402.18789*, 2024.

[43] J. Liu, Z. Wu, J.-W. Chung, F. Lai, M. Lee, and M. Chowdhury, "Andes: Defining and enhancing quality-of-experience in llm-based text streaming services," *arXiv preprint arXiv:2404.16283*, 2024.

[44] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[45] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, "Retrieval augmentation reduces hallucination in conversation," *arXiv preprint arXiv:2104.07567*, 2021.

[46] Y. Zhu, J.-C. Gu, C. Sikora, H. Ko, Y. Liu, C.-C. Lin, L. Shu, L. Luo, L. Meng, B. Liu *et al.*, "Accelerating inference of retrieval-augmented generation via sparse context selection," *arXiv preprint arXiv:2405.16178*, 2024.

[47] C. Jin, Z. Zhang, X. Jiang, F. Liu, X. Liu, X. Liu, and X. Jin, "Ragcache: Efficient knowledge caching for retrieval-augmented generation," *arXiv preprint arXiv:2404.12457*, 2024.

[48] J. Yao, H. Li, Y. Liu, S. Ray, Y. Cheng, Q. Zhang, K. Du, S. Lu, and J. Jiang, "Cacheblend: Fast large language model serving with cached knowledge fusion," *arXiv preprint arXiv:2405.16444*, 2024.

[49] J. Li, Y. Jiang, Y. Zhu, C. Wang, and H. Xu, "Accelerating distributed {MoE} training and inference with lina," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 945–959.

[50] J. Yao, Q. Anthony, A. Shafi, H. Subramoni, and D. K. D. Panda, "Exploiting inter-layer expert affinity for accelerating mixture-of-experts model inference," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2024, pp. 915–925.

[51] Z. Du, S. Li, Y. Wu, X. Jiang, J. Sun, Q. Zheng, Y. Wu, A. Li, H. Li, and Y. Chen, "Sida: Sparsity-inspired data-aware serving for efficient and scalable large mixture-of-experts models," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 224–238, 2024.

[52] L. Xue, Y. Fu, Z. Lu, L. Mai, and M. Marina, "Moe-infinity: Activation-aware expert offloading for efficient moe serving," *arXiv preprint arXiv:2401.14361*, 2024.

[53] K. Kamahori, Y. Gu, K. Zhu, and B. Kasikci, "Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models," *arXiv preprint arXiv:2402.07033*, 2024.

[54] H. Huang, N. Ardalani, A. Sun, L. Ke, H.-H. S. Lee, A. Sridhar, S. Bhosale, C.-J. Wu, and B. Lee, "Towards moe deployment: Mitigating inefficiencies in mixture-of-expert (moe) inference," *arXiv preprint arXiv:2303.06182*, 2023.

[55] Y. Sheng, S. Cao, D. Li, B. Zhu, Z. Li, D. Zhuo, J. E. Gonzalez, and I. Stoica, "Fairness in serving large language models," *arXiv preprint arXiv:2401.00588*, 2023.

[56] B. Li, Y. Jiang, V. Gadepally, and D. Tiwari, "Toward sustainable genai using generation directives for carbon-friendly large language model inference," *arXiv preprint arXiv:2403.12900*, 2024.

[57] K. Hong, G. Dai, J. Xu, Q. Mao, X. Li, J. Liu, Y. Dong, Y. Wang *et al.*, "Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 148–161, 2024.

[58] C. Lin, Z. Han, C. Zhang, Y. Yang, F. Yang, C. Chen, and L. Qiu, "Parrot: Efficient serving of llm-based applications with semantic variable," *arXiv preprint arXiv:2405.19888*, 2024.

[59] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, "Flashattention-3: Fast and accurate attention with asynchrony and low-precision," *arXiv preprint arXiv:2407.08608*, 2024.

[60] L. Chen, M. Zaharia, and J. Zou, "Frugalgpt: How to use large language models while reducing cost and improving performance," *arXiv preprint arXiv:2305.05176*, 2023.

[61] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, Z. Zhang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi *et al.*, "Specinfer: Accelerating large language model serving with tree-based speculative inference and verification," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 932–949.

[62] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica, "Routellm: Learning to route llms with preference data," *arXiv preprint arXiv:2406.18665*, 2024.