Advance in Efficient Large Language Models Serving Systems

Yunpeng Tai
School of Data Science
224045016@link.cuhk.edu.cn

***Abstract*—With the rapid development of artificial intelligence (AI), large language models (LLMs) have emerged to make life more convenient for people. However, the deployment of LLMs, encompassing numerous parameters, demands substantial computational power and high memory consumption. The computational characteristics of LLMs pose significant serving challenges in real-world scenarios that necessitate low latency and high throughput. This survey covers cutting-edge advancements in efficient LLMs serving systems, including low-bit quantization, parallel computing, and memory management. Additionally, this survey delves into prospective future directions in the design of efficient LLMs serving systems, offering insights to researchers striving to overcome the barriers of effective LLMs serving.**

***Index Terms*—large language models, serving system, quantization, parallel computing, memory management**

I. Introduction

Generative large language models (LLMs) have indeed become a cornerstone in the rapid evolution of artificial intelligence (AI), pushing the boundaries of what machines can achieve in understanding and interacting with human language. These models have not only excelled in traditional NLP tasks but have also opened up new avenues for AI applications:

- Machine Translation [1]: LLMs have significantly improved the accuracy and fluency of translations, making cross-cultural communication smoother and more accessible. They analyze vast amounts of bilingual text to learn context, idioms, and nuances, thereby reducing errors that were common in older translation systems.
- Sentiment Analysis [2]: By processing extensive datasets of human expressions, LLMs can now detect subtle emotional cues in text, helping businesses understand consumer sentiment at a granular level. This capability aids in brand management, product development, and customer service by offering insights into public perception.
- Question Answering [3]: With models like those from the GPT family, the ability to provide accurate and contextually relevant answers to complex queries has been enhanced. This technology is pivotal in educational tools, customer support, and information retrieval systems, where quick, precise answers are crucial.
- Text Generation [4]: From writing articles to creating poetry or scripts, LLMs can generate text that is increasingly difficult to distinguish from human-written content. This has implications for content creation, where AI can assist or even autonomously produce written material, reducing the workload on human writers.

The development of Transformer-based architectures like the GPT-family (Generative Pre-trained Transformer) [5]–[7], LLaMA-family [8]–[10], and other notable public LLMs such as OPT [11], BLOOM [12], Mistral [13], Baichuan [14], and GLM [15], [16] has been instrumental. These architectures have:

Revolutionized NLP: By introducing self-attention mechanisms, these models can weigh the importance of different words in a sentence, leading to better contextual understanding and generation of text. This shift has allowed for more dynamic and fluid processing of language, moving away from rigid rule-based systems to more adaptable, learning-based approaches. **Automated Programming** [17]: Assisting developers by suggesting code, auto-completing lines, and even writing small programs or scripts based on natural language instructions. **Science Discovery** [18]: Helping researchers by analyzing scientific literature, predicting outcomes, or even suggesting new research directions through pattern recognition in vast datasets. **Personalized Digital Assistants** [19]: Enhancing the personalization and predictive capabilities of digital assistants, making them more intuitive and responsive to user needs. **Medical Diagnosis** [20]: Aiding healthcare professionals in diagnosing diseases, recommending treatments, and analyzing patient data to improve outcomes.

The versatility of these models stems from their ability to learn from vast amounts of text data, which allows them to understand and manipulate language in ways previously thought impossible. Their impact is profound, not just within the tech industry but across various sectors, demonstrating the potential of AI to revolutionize how we interact with technology and each other. As these models continue to evolve, their integration into everyday applications will likely grow, further blurring the lines between human and machine interaction.

The unprecedented success of large language models (LLMs) in recent years has revolutionized various fields, from natural language understanding to automated content generation. However, this success comes with its own set of challenges, most notably their formidable computational requirements during serving.

1) Model Size and Complexity: LLMs often contain billions of parameters, which not only demand vast storage space but also require significant computational power to process each request. Their complexity arises from the intricate network architectures designed to capture deep

semantic understanding, making real-time processing a daunting task.

2) Energy Consumption: The training and serving of these models consume enormous amounts of energy. Data centers hosting these models contribute significantly to carbon footprints, raising environmental concerns. The energy required for each interaction with an LLM can be equivalent to the daily energy consumption of multiple households, highlighting the sustainability issues at play.
3) Scalability: Scaling these models to serve a large number of users simultaneously while maintaining performance efficiency is a significant challenge. Traditional scaling methods like vertical scaling (increasing the power of a single server) or horizontal scaling (adding more servers) face limitations due to the models' resource demands.
4) Accessibility: The high computational overhead means that only organizations with substantial financial and infrastructural resources can afford to deploy these models. This creates a digital divide, where only large corporations or well-funded research institutions can leverage the full potential of LLMs, leaving smaller entities or individuals at a disadvantage.

This survey paper focuses on addressing the critical need for efficient LLM serving. It delves into the multifaceted strategies proposed by the research community to tackle these challenges:

Low-bit Quantization [21]–[23]: This technique involves reducing the precision of the model's parameters from 32-bit or 16-bit to lower-bit representations like 8-bit or even 4-bit. By doing so, the model size is significantly reduced, allowing for faster computation and lower memory usage without substantially sacrificing accuracy.

Parallel Computing [24]–[26]: Strategies like model parallelism, where different parts of the model are processed on different computational units, and data parallelism, where a single model processes multiple data batches simultaneously, are explored. These methods help in distributing the computational load, thereby enhancing throughput and reducing latency.

Memory Management [27], [28]: Efficient memory management techniques, such as dynamic memory allocation, caching strategies, and optimized data loading, are crucial. These approaches aim to minimize the memory footprint during model inference, which is especially important for models that exceed the memory capacity of typical hardware.

The paper is structured as follows: we begin by providing background information on the challenges of serving LLMs in Section 2. Section 3 discusses the advancements in efficient LLM serving systems, focusing on low-bit quantization, parallel computing, and memory management, respectively. Section 4 predicts the future of efficient LLM serving systems, highlighting potential research directions and emerging trends. Finally, Section 5 concludes the paper by summarizing the key findings and insights presented in this survey.

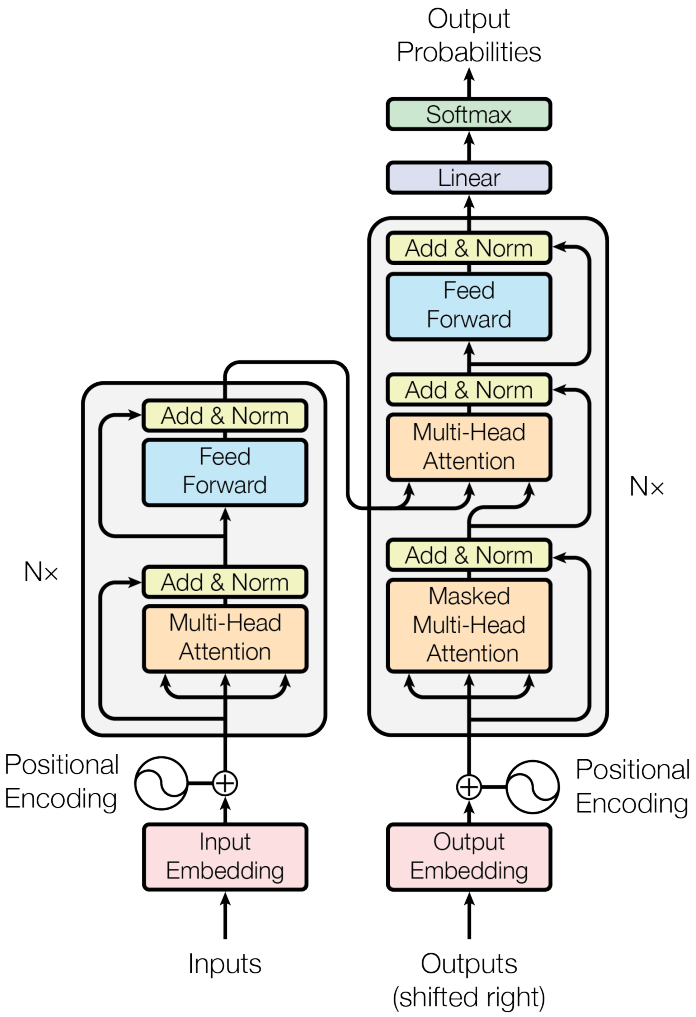


Fig. 1. The architecture of the Transformer model.

II. Background

A. The Transformer Architecture

Transformer-based Large Language Models (LLMs) have brought about a major change in the field of natural language processing by introducing a new approach to understanding and generating human language. At the core of this advancement lies the Transformer architecture [29]. As is shown in Figure 1, transformer utilizes self-attention mechanisms to determine the significance of various parts of the input data in prediction-making.

Mathematically, the self-attention mechanism in Transformers can be described as follows: Given an input sequence $X = [x_1, x_2, ..., x_n]$, the Transformer calculates queries Q, keys K, and values V through linear transformations of X. The self-attention scores are then computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \qquad (1)$$

where d_k is the dimension of the keys. This mechanism allows the model to focus on different parts of the input

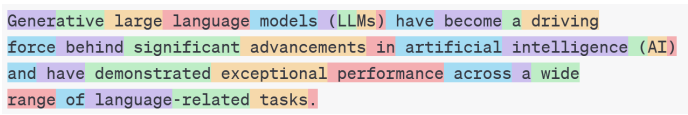


Fig. 2. The example text tokenization process of OpenAI's tokenizer.

sequence for each element of the output, capturing complex dependencies regardless of their distance in the input sequence.

Another crucial component of Transformers is the Feed-Forward Network (FFN), which is found in every layer of the Transformer and plays a key role in its computational complexity. The FFN usually includes two linear transformations separated by a non-linear activation function, commonly shown as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2, \tag{2}$$

Here, W_1, W_2, b_1, and b_2 represent the learnable parameters of the FFN. The non-linear function $\max(0, \cdot)$, also known as ReLU, adds necessary non-linearity to the model, enabling it to learn complex patterns.

The FFN contributes significantly to the model's parameters, affecting its memory usage and computational demands. In each Transformer layer, following the multi-head attention (MHA) step which gathers information from various parts of the input, the FFN independently processes this combined information for each position. This parallel processing ability is a key feature of the Transformer, allowing it to handle sequences efficiently. However, this also means that the computational load and memory requirements increase with the input sequence length and network depth.

The integration of self-attention and FFN in Transformer-based LLMs allows these models to effectively grasp diverse linguistic contexts and details, achieving impressive results in numerous NLP tasks. Nevertheless, the significant computational demands for both training and inference have become a major research focus, aiming to enhance efficiency without sacrificing performance. The Transformer model incorporates additional crucial elements such as position encoding, providing positional information for each token in the sequence, and the multi-head attention mechanism, enabling the model to attend to various sequence segments in different representational dimensions.

B. Auto-regressive Decoding of LLM Inference

During inference, LLMs first tokenize the input text into a sequence of tokens, which are then fed into the model for processing. Tokenization is the process of converting raw text into a sequence of tokens, where each token represents a word, subword, or character. Figure 2 illustrates an example of text tokenization using OpenAI's tokenizer.

LLM inference, particularly in models like GPT (Generative Pre-trained Transformer), often employs an auto-regressive decoding approach. This method is central to how these models generate text, ensuring that each new word or token produced takes into account the entire sequence generated so far. Given the input sequence $X = [x_1, x_2, ..., x_n]$, the model generates the next token x_{n+1} by conditioning on the entire sequence X: $P(x_{n+1}|X)$. This process is repeated iteratively to generate the desired output text. The auto-regressive nature of this decoding strategy allows the model to capture long-range dependencies and generate coherent text.

TABLE I
C4 VALIDATION PERPLEXITIES OF QUANTIZATION METHODS FOR DIFFERENT TRANSFORMER SIZES FROM 125M TO 13B PARAMETERS.

Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

III. The Recent Advances

A. Low-bit Quantization

This section delves into cutting-edge low-bit quantization techniques that allow for efficient representation of model weights and activations. By utilizing fewer bits (i.e., less than 32) to represent numerical values, these techniques greatly decrease memory usage and speed up inference on hardware platforms. One approach involves quantizing LLM, with quantization methods falling into two main categories: **Quantization-Aware Training** (QAT) and **Post-Training Quantization** (PTQ).

PTQ reduces the computational precision of model weights and activations after training, thereby reducing the model's memory footprint and computational requirements. This technique is particularly effective for fine-tuning pre-trained models on specific tasks, as it allows for faster inference without significant loss in accuracy. PTQ methods include uniform quantization, non-uniform quantization, and mixed-precision quantization, each with its trade-offs between model size, accuracy, and computational efficiency. LLM.int8() [22] is a recent PTQ method that quantizes model weights and activations to 8-bit integers, significantly reducing the model size and improving inference speed without compromising performance. As illustrated in Figure 3, when provided with 16-bit floating-point inputs $\mathbf{X}_{f16}$ and weights $\mathbf{W}_{f16}$, the features and weights are divided into sub-matrices of high-value features and other values. The high-value feature matrices are calculated in 16-bit, while all other values are calculated in 8-bit. The 8-bit vector-wise multiplication is conducted by scaling with the maximum absolute values of rows and columns in $\mathbf{C}_x$ and $\mathbf{C}_w$, followed by quantizing the results to Int8. The Int32 matrix multiplication results (Out32) are dequantized using the outer product of the normalization constants $\mathbf{C}_x \bigotimes \mathbf{C}_w$. Finally, the outputs from both high-value and regular calculations are accumulated in 16-bit floating point format. As presented

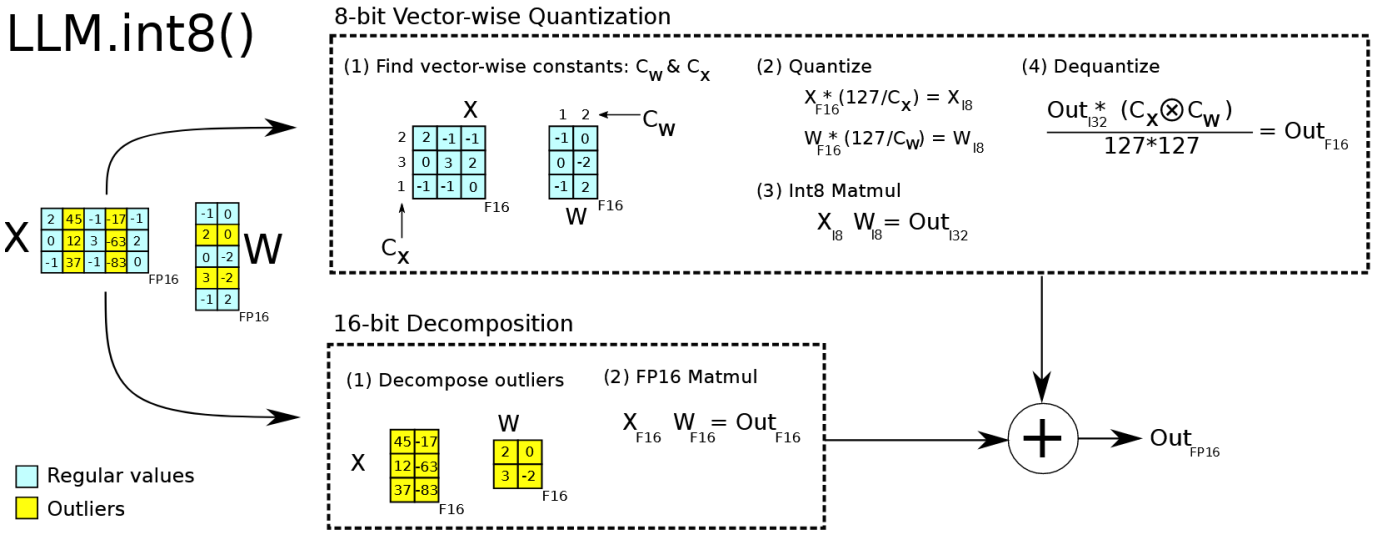


Fig. 3. The Schema of LLM.int8(). Given 16-bit floating-point inputs $\mathbf{X}_{f16}$ and weights $\mathbf{W}_{f16}$, the features and weights are decomposed into sub-matrices of large magnitude features and other values. The outlier feature matrices are multiplied in 16-bit. All other values are multiplied in 8-bit. We perform 8-bit vector-wise multiplication by scaling by row and column-wise absolute maximum of $\mathbf{C}_x$ and $\mathbf{C}_w$ and then quantizing the outputs to Int8. The Int32 matrix multiplication outputs Out32 are dequantization by the outer product of the normalization constants $\mathbf{C}_x \otimes \mathbf{C}_w$. Finally, both outlier and regular outputs are accumulated in 16-bit floating point outputs.

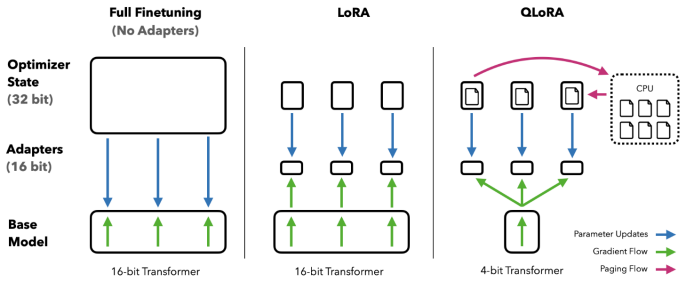


Fig. 4. Different finetuning methods and their memory requirements. QLORA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

in Table II-B, the C4 validation perplexities of quantization methods for various transformer sizes ranging from 125M to 13B parameters are displayed. It is observed that absmax, row-wise, zeropoint, and vector-wise quantization methods result in notable performance degradation as we increase the size, especially noticeable at the 13B parameter mark where the perplexity of 8-bit 13B is worse than that of 8-bit 6.7B. However, when utilizing LLM.int8(), the full perplexity is restored as we scale. Zeropoint quantization displays an advantage due to its asymmetric nature, but this advantage diminishes when used alongside mixed-precision decomposition.

On the other hand, QAT integrates quantization into the training process, enabling the model to learn with lower precision from the start. This approach ensures that the model adapts to the reduced precision during training, leading to better performance on low-bit hardware. QAT methods include quantization-aware backpropagation, which incorporates quantization errors into the training process, and dynamic quantization, which quantizes model weights and activations dynamically during training. Qlora [23] introduces multiple innovations designed to reduce memory use without sacrificing performance: (1) 4-bit NormalFloat, an information theoretically optimal quantization data type for normally distributed data that yields better empirical results than 4-bit Integers and 4-bit Floats. (2) Double Quantization, a method that quantizes the quantization constants, saving an average of about 0.37 bits per parameter (approximately 3 GB for a 65B model). (3) Paged Optimizers, using NVIDIA unified memory to avoid the gradient checkpointing memory spikes that occur when processing a mini-batch with a long sequence length. As displayed in Figure 4, Qlora combines these contributions into a better tuned LoRA approach that includes adapters at every network layer and thereby avoids almost all of the accuracy tradeoffs seen in prior work. Results which are shown in Table II suggest that 16-bit, 8-bit, and 4-bit adapter methods replicate the performance of the fully finetuned 16-bit baseline. This suggests that the performance lost due to the imprecise quantization can be fully recovered through adapter finetuning after quantization.

B. Parallel Computing

This section explores parallel computation strategies specifically designed for large language models. By utilizing the parallel processing capabilities of modern hardware architectures, these methods distribute computations across multiple cores or devices, resulting in significant speed improvements during inference.

Many strategies for **model parallelism** were initially introduced to support the distributed training of large-scale DNNs, particularly Transformer-based models. One such approach is tensor model parallelism (TP), which divides the model layers

TABLE II
EXPERIMENTS COMPARING 16-BIT BRAINFLOAT (BF16), 8-BIT INTEGER (INT8), 4-BIT FLOAT (FP4), AND 4-BIT NORMALFLOAT (NF4) ON GLUE AND SUPER-NATURALINSTRUCTIONS. QLORA REPLICATES 16-BIT LORA AND FULL-FINETUNING.

Dataset	GLUE (Acc.)	Super-NaturalInstructions (RougeL)				
Model	RoBERTa-large	T5-80M	T5-250M	T5-780M	T5-3B	T5-11B
BF16	88.6	40.1	42.1	48.0	54.3	62.0
BF16 replication	88.6	40.0	42.2	47.3	54.9	-
LoRA BF16	88.8	40.5	42.6	47.1	55.4	60.7
Qlora Int8	88.8	40.4	42.9	45.4	56.5	60.7
Qlora FP4	88.6	40.3	42.4	47.5	55.6	60.9
Qlora NF4 + DQ	-	40.4	42.7	47.7	55.3	60.9

(such as attention and FFN) into separate portions based on internal dimensions (like head and hidden), assigning each to a different device (such as a GPU). Megatron-LM [24] can greatly decrease inference latency by utilizing parallel computing, commonly employed across multiple GPUs on the same machine, particularly in situations with high-speed NVLink connections.

As shown in Figure 5, the PaLM architecture [30], implements a sophisticated 2D tensor parallelism strategy to effectively partition the layout for large-scale Transformer inference. This method is particularly noted for its lower theoretical communication complexity when deployed on clusters with more than 256 devices, significantly reducing the overhead associated with data transfer in large-scale distributed systems.

The primary goal here is to manage the vast computational demands of modern deep learning models. By employing this parallelism, PaLM aims to not only scale effectively but also to maintain performance efficiency across an expansive network of GPUs. This strategy is crucial for applications where computational resources are paramount, such as in training or inference of large language models or complex vision tasks.

- Attention and Feed-forward Layers: PaLM distributes both the attention and feed-forward layers across the hardware. This distribution ensures that each GPU handles a manageable portion of the workload, reducing bottlenecks and allowing for parallel computation which increases throughput.
- Data and Model Parallelism: The combination of data and model parallelism allows for a balance between keeping each GPU busy with unique data (data parallelism) and spreading the model's parameters across multiple devices (model parallelism). This dual approach optimizes memory usage as the entire model does not need to fit into the memory of a single GPU, and it also enhances computational efficiency by allowing for simultaneous processing of different parts of the model.
- Hybrid Tensor Partition Strategy: For scenarios where multi-query attention mechanisms are used, with only one head for keys and values, PaLM incorporates data parallelism into its hybrid tensor partition strategy. This approach ensures that even with fewer attention heads, the system can still leverage parallel processing to maintain high performance.

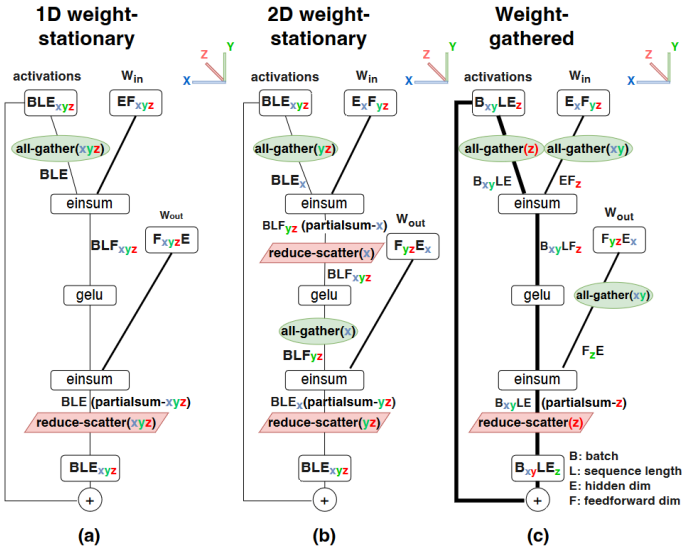
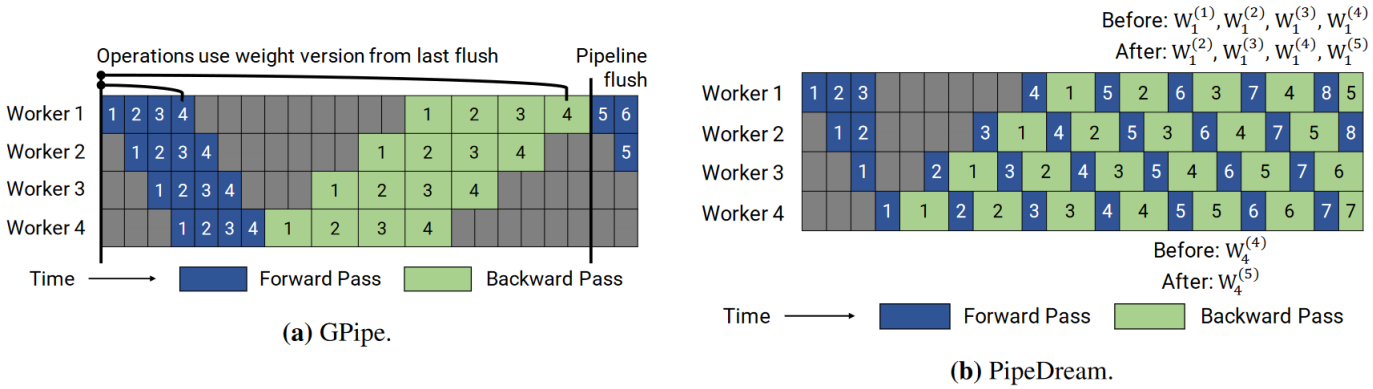


Fig. 5. Partitioning layouts for feedforward layer.

Pipeline model parallelism (PP), as discussed in [31], is a sophisticated technique aimed at optimizing the performance of large-scale neural network models by distributing their computational load across multiple devices. The primary goal of PP is to facilitate the training and inference of models that are too large to fit into the memory of a single device. By breaking down the model into manageable segments, PP allows for the efficient use of hardware resources, thereby reducing the time and computational costs associated with training complex models.

As illustrated in Figure 6, the layers of the model are arranged in a sequential pipeline across several devices when applying pipeline parallelism. **1) Layer Distribution**: Each



(a) GPipe.

(b) PipeDream.

Fig. 6. Timelines of different pipeline-parallel executions. Without loss of generality, forward and backward passes are assumed to take twice as long as forward passes; forward passes are shown in blue and backward passes are shown in green. Numbers indicate microbatch ID, time is shown along x-axis, per-worker utilization is shown along the y-axis. GPipe maintains a single weight version, but periodically flushes the pipeline. PipeDream does not introduce periodic pipeline flushes, but maintains multiple weight versions.

layer of the model is assigned to a specific device in the sequence. For example, if a model has 10 layers and there are 3 devices, each device could handle roughly 3 to 4 layers, depending on the complexity and memory requirements of each layer. **2) Pipeline Stages**: Each device handles what is known as a 'pipeline stage'. A stage includes multiple consecutive layers of the model. The number of layers in each stage can be adjusted based on the device's capacity or the specific needs of the workload. **3) Data Flow**: Data flows through this pipeline in a manner similar to an assembly line. When an input enters the first stage, it undergoes processing by the layers on the first device. After processing, the output is passed to the next device for further processing, and this continues until the data has passed through all stages. **4) Synchronization**: To maintain efficiency, synchronization points are introduced to ensure that data is processed in a coordinated manner. This might involve techniques like bubble scheduling or pipeline flushing to manage the potential idle times between stages. **5) Background and Context**: The necessity for pipeline parallelism arises from the ever-increasing size of neural networks, especially in applications like natural language processing (NLP), where models with billions of parameters are common. Traditional data parallelism, where the same model is replicated across devices, becomes less efficient with such large models due to memory constraints and communication overheads.

By employing pipeline model parallelism, we can not only manage the memory constraints of large models but also utilize hardware resources more effectively. This technique not only speeds up the training process but also allows for the deployment of models that would otherwise be impractical to run on single devices. Thus, understanding and implementing pipeline parallelism is crucial for advancing the capabilities of deep learning in various high-demand sectors.

However, while Pipeline Parallelism excels at boosting throughput, it does not inherently reduce the time required to process a single input from start to finish, known as latency. Each input still goes through all the stages of the pipeline, meaning that the time for a single piece of data to traverse the entire pipeline remains the same. In contrast, Tensor Parallelism (TP) focuses on distributing the workload across multiple processing units or devices in such a way that each unit handles a part of the computation for every input simultaneously. This method can indeed decrease latency because it essentially parallelizes the computation within a single input, allowing for quicker processing of individual data points.

Sequence parallelism (SP) [32] offers numerous innovative approaches and configurations tailored specifically for enhancing large language model (LLM) inference. The core concept behind SP in the context of LLM inference revolves around optimizing the distribution of computational tasks and storage demands. SP distributes the computational and storage load by dividing the processing of long sequences across multiple GPUs. This distribution is not random; it follows the sequence length dimension, meaning each GPU processes a segment of the sequence. This approach significantly reduces the memory footprint on individual GPUs, allowing for the handling of much longer sequences than would be possible with a single GPU setup. By splitting the sequence, SP enables: **Increased Sequence Length**: Allowing for the analysis and generation of longer texts or data streams, which is crucial for tasks like document summarization, long-form text generation, or any application requiring understanding or generating extended contexts. **Efficiency**: It enhances the efficiency of GPU usage by ensuring that computational resources are not bottlenecked by memory limitations, thus improving throughput and reducing processing time.

As displayed in Figure 7, implementing SP via a ring of hosts allows LLMs to scale to handle much larger data sets or documents, which is essential in fields like legal analysis, scientific research, or any area requiring deep textual analysis. By reducing memory bottlenecks, SP can significantly speed up inference times, making real-time applications of LLMs more feasible.

Fig. 7. Top (a): Ring Attention uses the same model architecture as the original Transformer but reorganize the compute. In the diagram, we explain this by showing that in a ring of hosts, each host holds one query block, and key-value blocks traverse through a ring of hosts for attention and feedforward computations in a block-by-block fashion. As we compute attention, each host sends key-value blocks to the next host while receives key-value blocks from the preceding host. The communication is overlapped with the computation of blockwise attention and feedforward. Bottom (b): Ring Attention computes the original Transformer block-by-block. Each host is responsible for one iteration of the query's outer loop, while the key-value blocks rotate among the hosts. As visualized, a device starts with the first query block on the left; then we iterate over the key-value blocks sequence positioned horizontally. The query block, combined with the key-value blocks, are used to compute self-attention (yellow box), whose output is pass to feedforward network (cyan box)

C. Memory Management

Efficient memory management is a major challenge in large language model (LLM) serving, especially due to the memory-intensive nature of transformer architectures. The KV cache's memory footprint is key for optimization when compared to model weights and workspace for other activations, particularly with the increasing demand for long-sequence inference. During incremental decoding, the KV cache memory fluctuates unpredictably. The traditional approach, like FasterTransformer, assumes a maximum sequence length and pre-allocates a continuous memory block, which leads to significant memory wastage for input batches with varying lengths and complex decoding scenarios generating multiple output sequences simultaneously, such as beam search and parallel decoding.

vLLM [27] introduces an innovative approach known as

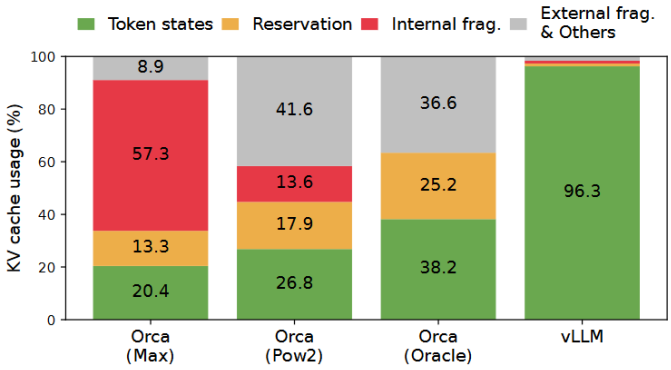


Fig. 8. Average percentage of memory wastes in different LLM serving systems.

paged attention, which revolutionizes how memory is managed in the context of large-scale language models. Traditionally, the KV (Key-Value) cache in these models has been stored in contiguous memory blocks, which can lead to inefficiencies, especially when dealing with variable-length sequences or when the model needs to handle multiple requests in parallel. Paged attention tackles these issues by partitioning the KV cache into non-contiguous memory blocks.

The advantages of paged attention are as follows: **1) Reduces Memory Fragmentation**: As shown in Figure 8, by allowing memory to be allocated in smaller, more manageable chunks, paged attention minimizes the problem of memory fragmentation, where free memory is broken into small, unusable pieces due to varying lengths of sequences. **2) Increases Batch Size**: With the ability to manage memory more efficiently, vLLM can support larger batch sizes. This means that more sequences can be processed in parallel, significantly speeding up the training and inference processes, which is critical for applications requiring real-time responses or handling large datasets. **3) Improves Throughput**: The non-contiguous memory allocation enables better utilization of hardware resources, leading to an increase in throughput. This improvement is vital for scenarios where high performance is necessary, such as in online services or real-time data processing. The implementation of paged attention not only enhances the efficiency of memory usage but also has broader implications for the scalability and performance of deep learning models. It allows for more dynamic handling of data, making models more adaptable to the varying demands of modern applications.

As shown in Figure 9, SpecInfer [28] suggests using tree attention and depth-first tree traversal to reduce unnecessary KV cache allocation for multiple output sequences that have a common prefix. SpecInfer's speculative inference and token tree verification offer two main advantages compared to the incremental decoding method used in current LLM inference systems.

Reducing memory accesses to LLM parameters is crucial for improving the performance of LLM inference. In the current incremental decoding method, generating a single token requires accessing all LLM parameters, which poses a challenge, especially for offloading-based LLM inference systems. These systems utilize limited computational resources, like a single commodity GPU, by offloading tasks to CPU DRAM and storage for parameter storage and retrieval. SpecInfer, on the other hand, reduces accesses to LLM parameters significantly when there is an overlap between speculated and actual token output. This reduction in memory accesses can lead to lower energy consumption by minimizing data transfers between GPU and CPU memory, as accessing GPU HBM consumes significantly more energy than arithmetic operations.

Serving large language models (LLMs) often faces delays in processing requests. For instance, the GPT-3 model, with its 175 billion parameters, can take several seconds to generate a response. To address this issue, the current method of incremental decoding relies on sequential dependencies between tokens, causing each token's computation to be influenced by all previously generated tokens. As a result, modern LLM serving systems must handle requests by generating tokens one by one. In SpecInfer, LLMs are employed to verify a speculated token tree in a single pass, allowing for simultaneous examination of all tokens in the tree. This new approach enables parallel processing of different tokens within a single request, ultimately reducing the overall latency of the LLM's inference process.

IV. Future Development

In the realm of low-bit quantization for large language models (LLMs), exploring more stable quantization methods is crucial due to the diverse scales at which these models operate. Quantization, which involves reducing the precision of the numerical representations used in model weights, is essential for deploying LLMs on devices with limited computational resources. Adopting quantization methods that align with the scaling law of LLMs ensures that as models grow in size and complexity, their performance does not degrade unexpectedly. This alignment means that the quantization process respects the inherent relationships and dependencies within the model's architecture, thereby maintaining the model's accuracy and efficiency even at lower bit depths. The purpose of this approach is to enable broader application of LLMs, from edge devices to cloud services, without significantly sacrificing performance.

Moving to parallel computation, the challenge often lies in managing the latency caused by inter-process communication. This latency can bottleneck the overall speed of computation, particularly in distributed systems where different parts of the model or data are processed on separate nodes or devices. To address this, strategies like optimized data routing, advanced synchronization techniques, or even hardware-specific optimizations can be employed. By reducing communication overhead, we can significantly decrease the time required for model inference or training, thereby speeding up the computation process. This enhancement is not just about speed

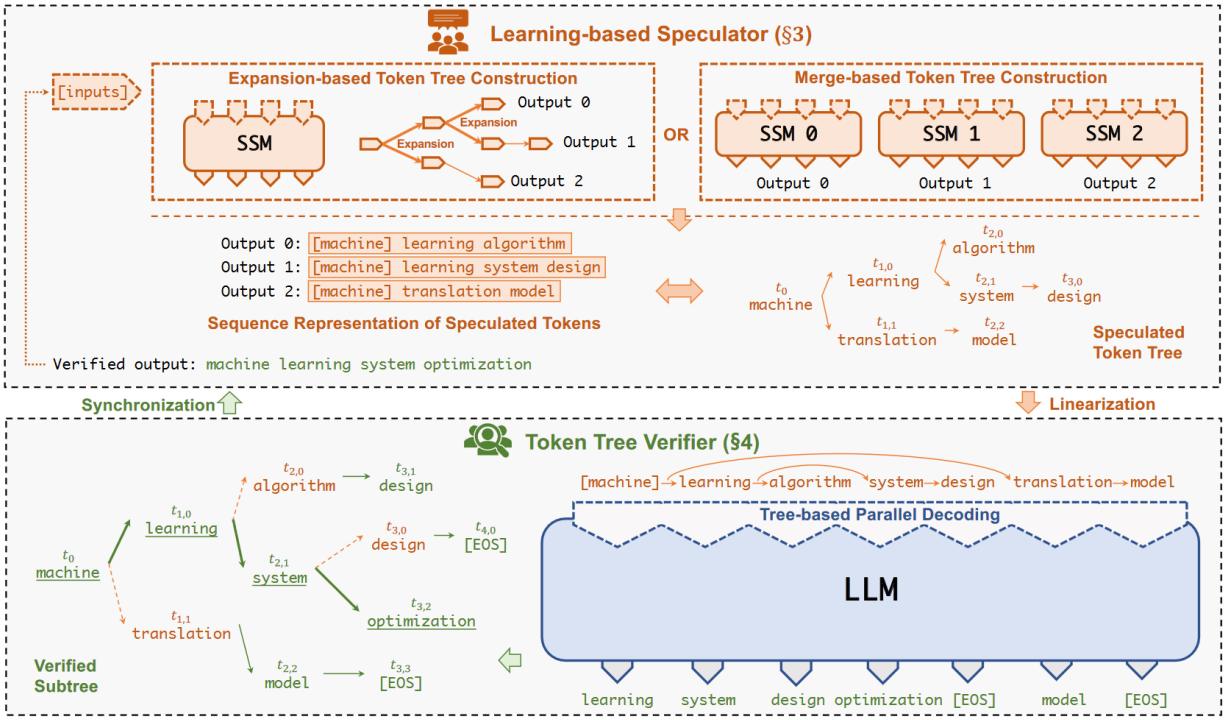


Fig. 9. An overview of SpecInfer's tree-based speculative inference and verification mechanism

but also about scalability, allowing LLMs to handle real-time processing tasks more effectively.

Regarding memory management, fine-grained memory strategies are often employed to optimize the use of available memory resources. However, these strategies can sometimes lead to performance degradation due to the increased overhead of managing small chunks of memory. To counteract this, one could look into more coarse-grained approaches or hybrid strategies that balance between memory efficiency and performance. For instance, implementing techniques like memory pooling or using memory-efficient data structures might help in reducing the overhead while still maintaining efficient memory usage. The goal here is to refine memory management techniques so that they do not impede the computational efficiency of LLMs, allowing for larger models to be run on devices with constrained memory without a substantial drop in performance.

In summary, by addressing these three core areas: **low-bit quantization**, **parallel computation**, and **memory management**, we aim to enhance the deployment and operational efficiency of large language models. Each improvement contributes to making LLMs more accessible, faster, and more resource-efficient, thereby broadening their applicability and utility across various computational environments.

V. Conclusion

In conclusion, the rapid advancement of artificial intelligence (AI) and the emergence of large language models (LLMs) have significantly improved convenience in people's lives. However, the substantial computational power and memory demands associated with LLM deployment present considerable challenges in real-world applications, particularly in achieving low latency and high throughput. This survey has explored the latest developments in efficient LLM serving systems, including approaches such as low-bit quantization, parallel computing, and memory management. Furthermore, it has highlighted prospective future directions in designing efficient LLM serving systems. These insights aim to guide researchers in addressing the key barriers to effective and efficient deployment of LLMs in practical scenarios.

Acknowledgment

This report documents the project I completed for the Advanced Operating Systems (CSC 6032) course. I want to extend my heartfelt thanks to Professor Yeh-Ching Chung for his guidance and support during the project. I also appreciate the assistance and feedback provided by Teaching Assistant ShiHao Hong, which was crucial in completing this work.

References

[1] S. Yang, Y. Wang, and X. Chu, "A survey of deep learning techniques for neural machine translation," *arXiv preprint arXiv:2002.07526*, 2020.

[2] S. Kumar, P. P. Roy, D. P. Dogra, and B.-G. Kim, "A comprehensive review on sentiment analysis: Tasks, approaches and applications," *arXiv preprint arXiv:2311.11250*, 2023.

[3] H. A. Pandya and B. S. Bhatt, "Question answering survey: Directions, challenges, datasets, evaluation matrices," *arXiv preprint arXiv:2112.03572*, 2021.

[4] J. Li, T. Tang, W. X. Zhao, J.-Y. Nie, and J.-R. Wen, "Pre-trained language models for text generation: A survey," *ACM Computing Surveys*, vol. 56, no. 9, pp. 1–39, 2024.

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. 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. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. teusz Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *ArXiv*, vol. abs/2005.14165, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:218971783

[7] 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.

[8] 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.

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

[10] 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.

[11] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.

[12] B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon *et al.*, "Bloom: A 176b-parameter open-access multilingual language model," *arXiv preprint arXiv:2211.05100*, 2022.

[13] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.

[14] A. Yang, B. Xiao, B. Wang, B. Zhang, C. Bian, C. Yin, C. Lv, D. Pan, D. Wang, D. Yan *et al.*, "Baichuan 2: Open large-scale language models," *arXiv preprint arXiv:2309.10305*, 2023.

[15] Z. Du, Y. Qian, X. Liu, M. Ding, J. Qiu, Z. Yang, and J. Tang, "Glm: General language model pretraining with autoregressive blank infilling," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2022, pp. 320–335.

[16] T. GLM, A. Zeng, B. Xu, B. Wang, C. Zhang, D. Yin, D. Zhang, D. Rojas, G. Feng, H. Zhao *et al.*, "Chatglm: A family of large language models from glm-130b to glm-4 all tools," *arXiv preprint arXiv:2406.12793*, 2024.

[17] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[18] "The promise and peril of generative ai," *Nature*, vol. 614, no. 1, pp. 214–216, 2023.

[19] X. L. Dong, S. Moon, Y. E. Xu, K. Malik, and Z. Yu, "Towards next-generation intelligent assistants leveraging llm techniques," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023, pp. 5792–5793.

[20] L. Li, J. Zhou, Z. Gao, W. Hua, L. Fan, H. Yu, L. Hagen, Y. Zhang, T. L. Assimes, L. Hemphill *et al.*, "A scoping review of using large language models (llms) to investigate electronic health records (ehrs)," *arXiv preprint arXiv:2405.03066*, 2024.

[21] Z. Yao, X. Wu, C. Li, S. Youn, and Y. He, "Zeroquant-v2: Exploring post-training quantization in llms from comprehensive study to low rank compensation," *arXiv preprint arXiv:2303.08302*, 2023.

[22] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale," *Advances in Neural Information Processing Systems*, vol. 35, pp. 30 318–30 332, 2022.

[23] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[24] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *arXiv preprint arXiv:1909.08053*, 2019.

[25] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.

[26] "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[27] 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.

[28] 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 generative large language model serving with tree-based speculative inference and verification," *arXiv preprint arXiv:2305.09781*, 2023.

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

[30] 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.

[31] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*. PMLR, 2021, pp. 7937–7947.

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