A Background Survey on Optimizing Memory Usage for LLM Training and Inference

Warren Low
e1115618@u.nus.edu

National University of Singapore

Abstract

Recently, large language models (LLMs), particularly those from the open-source community, have scaled to unprecedented numbers of parameters. However, this rapid growth poses significant challenges for consumer-grade GPUs, which often lack the memory capacity to handle the training or inference of such models. In this paper, we survey state-of-the-art techniques aimed at reducing memory consumption in both training and inference, making large models more accessible. We begin by quantifying memory requirements, then explore a wide range of optimization methods, including quantization techniques, low-rank adaptation (LoRA) and its variants, attention-based strategies, and specialized techniques for training and inference. Additionally, we evaluate the limitations and trade-offs associated with these memory optimization strategies.

1 Introduction

In recent years, the field of Large Language Models (LLMs) has seen unprecedented growth, driven by scaling laws such as those proposed by OpenAI and Google Deepmind [23, 20]. These laws demonstrate that increasing model size correlates strongly with improved performance across various benchmarks, embodying the principle that "scale is all you need." This trend is exemplified by open-source models like Qwen2-72B [42] and LLaMA 3 405B [15], which push the boundaries of model size and capability.
However, this rapid scaling comes at a significant cost to accessibility. The exponential growth in memory requirements for these larger models far exceeds the capacity of most consumer-grade Graphic Processing Units (GPUs), creating a substantial barrier to entry for many researchers and developers. This survey paper addresses this challenge by examining a range of methods designed to reduce memory costs in both training and inference scenarios. We explore techniques including quantization, Low-Rank Adaptation (LoRA), attention optimization, pruning, and alternative LLM architectures.
Our survey builds upon and extends previous works [50, 8, 18] that have examined memory compression for LLMs and related challenges. However, our paper distinguishes itself in several key ways:

1. We provide a comprehensive overview of memory optimization techniques for both training and inference within a single paper, offering a holistic view of the field.
2. We include an up-to-date analysis of recent innovations, addressing the surge in novel research since previous surveys.
3. We offer a detailed breakdown of LLM memory costs for both training and inference, providing a quantitative foundation for understanding the scale of the challenge.

By combining these elements, we aim to provide a "kaleidoscopic lens" on the topic of memory optimization for LLMs, offering both a broad perspective and detailed insights into specific techniques.
Moreover, we examine the challenges faced by researchers in this field and explore current directions for improving memory optimization in both inference and training. This includes addressing issues such as the lack of standardized benchmarks, the growing need for long-context memory management, and the emerging challenges posed by optimizing memory for multimodal models.
Through this comprehensive survey, we seek to educate readers on existing methods for compressing GPU memory usage during fine-tuning and inference, while also providing a quantitative understanding of LLM memory requirements. Our goal is to make large language models more accessible to a broader range of researchers and practitioners, potentially accelerating innovation in this rapidly evolving field.

1.1 Calculating Memory Requirements

A key point missed out by previous survey papers is the itemized quantification of the scale of memory usage by large language models. Zhou et al. and Chavan et al. [8, 50] quantifies the sheer scale of GPU VRAM to finetune LLMs such as LLaMA, and also covers some of the techniques listed here. However, they primarily focus on optimizing both speed and memory costs during inference, while our survey covers memory costs for both inference and training.

1.1.1 Memory Required for Inference

Generally, there are multiple components that use GPU memory. Firstly, the framework used takes up memory to preload GPU kernels. For example, PyTorch loads 300MB-2GB of memory for CUDA kernels. On top of that, we have to load the parameters into the GPU. This is quantified by:

$$\text{VRAM for parameters (GB)} = \frac{\text{No. of Parameters} \times \text{Precision}}{8 \times 10^9} \tag{1}$$

Thus, if we were running an 8B model on FP32, we would need 8B * (32/8) / 1e9 GB of VRAM, which equates to 32GB of VRAM just for the parameters. Additionally, for this paper, we estimate that 20% of the VRAM used for parameters will be used for activations, the intermediate outputs within the model.
However, a significant component of memory usage during inference is the KV cache. According to Zhang et al. [45], the size of the KV cache can be calculated as:

$$\text{KV Cache Size} = b \times n \times l \times 2 \times h \times c \tag{2}$$

where b is the batch size, n is the number of tokens in each sequence, l is the number of layers in the model, 2 is for key and value, h is the number of key/value attention heads, and c is the number of channels in a single head of key/value activation embedding. Combining these components, we can estimate the total VRAM required for inference as:

$$\text{Total VRAM} = \text{VRAM for parameters} + \text{VRAM for activations} + \text{KV Cache Size} + \text{CUDA kernels} \tag{3}$$

This leads to a more comprehensive estimate of VRAM usage during inference. For example, for an 8B parameter model on FP16, with typical values for other parameters, the total VRAM usage could easily exceed 40GB, depending on the specific model architecture and inference settings.
We ignore the output during inference VRAM calculation as this varies with different transformers. This calculation underscores the significant memory demands of large language models during inference, highlighting the need for memory optimization techniques.

1.1.2 Memory Required for Training

Similar to inference, we have to preload CUDA kernels, which again take 300MB to 2GB of VRAM. On top of this, we now have to store the parameters again, using the same formula above. Additionally, we will also need to load the gradients into the GPU during backpropagation, which is the same size as the parameters. The activations themselves take up around the same VRAM as the parameters for weights during training. Furthermore, the optimizer states (e.g., for AdamW, the most common optimizer used for LLM training) take up 8 bytes per parameter as it stores two states. Thus we can estimate, for a 8B model:

$$\text{Total VRAM (GB)} = \frac{8 \times 8}{10^9} + 3 \times \frac{8 \times 4}{10^9} = 160 \text{ GB} \tag{4}$$

This calculation shows that approximately 160GB of VRAM is required just to natively finetune an 8B parameter LLM.

2 Preliminaries

2.1 Large Language Models (LLMs)

Large Language Models (LLMs) are advanced AI systems trained on vast corpora of text data to predict the next token in a sequence. Typically based on or derived from the Transformer architecture introduced by Vaswani et al. [40] from Google Brain in 2017, LLMs consist of multiple layers, each containing attention blocks, Feed-Forward Network (FFN) layers, and Layer Normalization (LayerNorm). These components are interconnected with residual connections, crucial for maintaining gradient flow during training. The attention mechanism, a key innovation of the Transformer architecture, is implemented as multi-head attention, allowing the model to focus on different aspects of the input simultaneously. For each attention head, three matrices are computed through linear projections: Query (Q), Key (K), and Value (V). These matrices are used to calculate attention scores and produce contextualized representations of the input sequence. The multi-head structure enables the model to capture various types of dependencies and relationships within the data, contributing to the LLM's ability to understand and generate human-like text across a wide range of tasks and domains.

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^Q \tag{5}$$

$$\mathbf{K}_i = \mathbf{X}\mathbf{W}_i^K \tag{6}$$

$$\mathbf{V}_i = \mathbf{X}\mathbf{W}_i^V \tag{7}$$

where $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$ are the Query, Key, and Value matrices for the i-th attention head, $\mathbf{X}$ is the input, and $\mathbf{W}_i^Q$, $\mathbf{W}_i^K$, and $\mathbf{W}_i^V$ are learnable weight matrices.
The attention output is then computed as:

$$\text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \text{Softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{d_k}}\right)\mathbf{V}_i \tag{8}$$

where d_k is the dimensionality of the key vectors.

2.1.1 Memory Consumption in Training and Inference

The memory footprint of LLMs varies significantly between training and inference phases. During training, memory is consumed by several components: the model parameters themselves, which represent the learned weights of the network; activations, which are the intermediate outputs of each layer; gradients, necessary for backpropagation and weight updates; and optimizer states, which store information for updating the model parameters efficiently. This combination of elements leads to substantial memory requirements, often limiting the size of models that can be trained on a single GPU. In contrast, the inference phase has a somewhat reduced memory footprint. Here, the main consumers are the model parameters and activations, similar to training, but without the need for storing gradients or optimizer states. However, inference introduces a new memory-intensive component: the KV cache (key-value cache). This cache stores precomputed key and value tensors for previously processed tokens, significantly speeding up autoregressive generation but potentially consuming large amounts of memory, especially for long sequences. The differing memory profiles of training and inference highlight the need for specialized optimization strategies for each phase of the LLM lifecycle.

2.2 KV Cache

The KV cache is a technique used to accelerate inference by storing previously computed key and value matrices. This prevents redundant computations across multiple decoding steps. The process works as follows:

1. During the initial (prefill) phase, all key and value matrices are computed and cached.
2. In subsequent decoding steps, the model retrieves the cached keys and values.
3. Only the keys and values for the new tokens need to be computed and appended to the cache.

This technique significantly reduces computation time but increases memory usage, especially for long sequences.

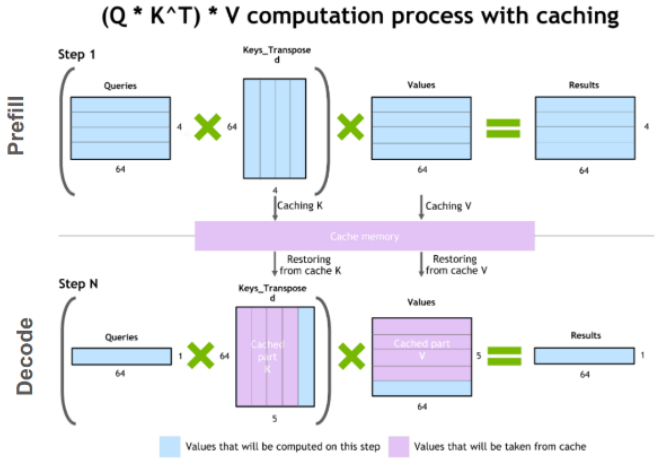

Figure 1: Illustration of caching, taken from the NVIDIA blog [32]

Newer transformers, such as LLaMA 3.1 [15], have experimented with different attention formats, such as Grouped-Query Attention [3] to help reduce memory costs, while others have tried training smaller language models which can outperform larger models like Phi. [2]. These help reduce memory costs, as well as improve inference speed through different perspectives. In this paper, we will discuss mainly memory compression techniques and attention optimization strategies. Different architectural designs such as Differential Transformers, RWKV and SSMs, [44, 33, 19] distillation methods, and data quality will not be discussed in the main portion of the paper, but will be further discussed in the appendix.

3 General Memory Optimization Techniques

This section examines various memory optimization strategies that significantly reduce GPU memory consumption for both training and inference of Large Language Models (LLMs). We focus on three primary techniques: quantization, [16, 46, 28, 41, 31] Low-Rank Adaptation (LoRA), [21, 49, 47, 43, 14] and attention-based optimizations. [37, 3, 12, 4]
Table 1 shows an overview of all methods discussed in this section.

Table 1: Comparison of General Memory Optimization Techniques for LLMs

Method	Papers	Pros	Cons
Quantization	GPTQ [16] QQQ [46] EXLLaMAV2 [39] LLM-QAT [28] BitNet [41, 31]	Significant memory reduction Potential speed increase No retraining for PTQ methods Adaptable during training for QAT	Potential accuracy loss Implementation complexity Hardware-specific optimizations needed Trade-off between compression and performance
Low-Rank Adaptation (LoRA)	LoRA [21] GaLORE [49] DoRA [43] QLoRA [14]	Efficient fine-tuning Reduced parameter count Preserves pretrained model Adaptable to various tasks	Most methods limited to fine-tuning Potential performance trade-offs Hyperparameter sensitivity Increased implementation complexity for variants
Attention Optimization	MQA [37] GQA [3] Flash Attention [12] Sliding Window Attention [4]	Reduces memory usage in key component Can improve inference speed Enables processing of longer sequences Hardware-efficient implementations possible	May affect model quality Often requires architectural changes Potential loss of long-range dependencies Increased implementation complexity

3.1 Quantization

Quantization is a technique that reduces the precision of model parameters and computations, thereby decreasing memory usage and potentially accelerating inference.

3.1.1 Principle of Quantization

Traditionally, LLMs operate using 32-bit floating-point (FP32) or 16-bit floating-point (FP16) precision. However, the memory footprint of these models scales linearly with the number of bits used per parameter. Table 2 shows a list of commonly used bit widths for training and inference.

Table 2: Comparison of Quantization Formats

Format	Bytes/Parameter	Range	Precision
FP32	4	$\pm 3.4 \times 10^{38}$	23 bits
BF16	2	$\pm 3.4 \times 10^{38}$	7 bits
FP16	2	$\pm 65,504$	10 bits
INT8	1	-128 to 127	8 bits
INT4	0.5	-8 to 7	4 bits

By reducing the number of bits used to represent each parameter, we can significantly decrease the model's memory footprint. For example, quantizing from FP32 to INT4 can potentially reduce memory usage by a factor of 8.

3.1.2 Challenges of Quantization

While quantization offers substantial memory savings, it introduces challenges:

1. **Quantization Error**: The reduction in precision can lead to a loss of information, potentially degrading model performance.
2. **Numerical Instability**: Especially during training, lower precision can cause numerical instabilities that affect model convergence.

Recent advancements in quantization techniques have focused on mitigating these issues, enabling more stable training and inference at lower precisions.

3.1.3 Types of Quantization

Quantization techniques can be broadly categorized based on their application in the LLM lifecycle:

- **Post-Training Quantization (PTQ)**: Applied after model training, primarily for inference optimization.
- **Quantization-Aware Training (QAT)**: Incorporates quantization effects during the training process, aiming to minimize performance loss.
- **Mixed Precision Training**: Uses different precisions for different parts of the model or different stages of computation during training.

The choice of quantization technique depends on the specific use case, with PTQ and QAT typically used for inference optimization, while mixed precision training is employed to reduce memory costs during the training phase.
In the following subsections, we will delve deeper into each of these quantization approaches, examining their methodologies, advantages, and limitations in the context of LLMs.

3.2 Quantization Techniques

Quantization can be applied at different stages of the LLM lifecycle. We discuss four main approaches: Post-Training Quantization, Quantization-Aware Training, Mixed Precision Training, and Fully Quantized Training.

3.2.1 Post-Training Quantization (PTQ)

Post-Training Quantization is applied after a model has been fully trained, reducing the size of an existing model without retraining. This approach offers significant advantages in terms of time and resource efficiency, as it doesn't require access to the original training data or extensive computational power. However, PTQ can lead to substantial accuracy drops, especially when using lower bit-widths, and its effectiveness varies across different model architectures.
A notable PTQ method is GPTQ [16], which mitigates some of these issues by analyzing each layer separately and using second-order information to predict and minimize quantization effects. While GPTQ enables faster model execution on fewer GPUs, it focuses solely on weight quantization, neglecting activations. Recent improvements have addressed this limitation: QQQ [46] extends GPTQ's applicability to fine-tuning scenarios by applying it to transferred weights, while EXLLaMAV2 [39] enhances performance through custom GPU code and variable quantization levels based on layer importance, offering a more nuanced approach to the trade-off between model size and accuracy.

3.2.2 Quantization-Aware Training (QAT)

Quantization-Aware Training incorporates quantization effects during the training process, allowing the model to adapt to quantization errors as it learns. This method generally achieves better accuracy than PTQ, especially for aggressive quantization, but at the cost of increased training time and computational resources. QAT also provides more flexibility in optimizing for specific hardware targets.
The LLM-QAT method [39] introduced QAT for large language models by simulating quantization during training and focusing on key model components like attention mechanisms and feed-forward networks. This targeted approach helps maintain model performance while achieving significant compression. Building on this, PEQA [24] further improved efficiency by selectively applying quantization to specific model parts, demonstrating that not all components require the same level of precision. This selective quantization strategy offers a more balanced approach between model size reduction and performance preservation.

3.2.3 Mixed Precision Training

Mixed Precision Training [30] strikes a balance between memory efficiency and accuracy by using lower precision (e.g., FP16) for most calculations while maintaining a high-precision (FP32) weight copy. This approach accelerates calculations and reduces memory usage during training, making it widely adopted in modern deep learning frameworks. However, the memory reduction is limited compared to more aggressive quantization techniques due to the necessity of keeping the FP32 weight copy.
The main advantage of mixed precision training is its ability to speed up training without significant accuracy loss, making it a popular choice for large-scale model training. However, it requires careful management of numeric stability, often through techniques like loss scaling, and may not be suitable for all model architectures or hardware configurations.

3.2.4 Fully Quantized Training (FQT)

Fully Quantized Training [17] is an emerging approach that pushes the boundaries of quantization by applying it to all aspects of the training process: weights, activations, gradients, and optimizer states. FQT offers greater memory savings than Mixed Precision Training and, unlike QAT, accelerates both training and inference. This comprehensive approach to quantization can potentially enable training of larger models on limited hardware or significantly speed up training of existing model sizes.
However, FQT introduces significant challenges in maintaining training stability and model performance. It requires specialized techniques to address issues like gradient underflow and overflow, as well as careful tuning of quantization parameters. While promising, FQT is still an active area of research, and its benefits and limitations in the context of LLMs require further investigation across various model sizes and tasks.

3.2.5 BitNet: Pushing Quantization to the Extreme

BitNet [41, 31] represents a significant advancement in quantization research, exploring the feasibility of 1-bit quantization for LLMs. This extreme approach offers the potential for massive memory reduction and computational efficiency but comes with substantial challenges in maintaining model performance.
The original BitNet paper [41] introduced a novel approach for training LLMs with binary (-1 or +1) weights and activations, while maintaining high-precision optimizer states and gradients. This was achieved by implementing a BitLinear layer in key components of the transformer architecture. While this approach demonstrated the potential for ultra-low precision LLMs, it faced challenges in reproducing results across different model sizes and tasks.
Addressing these limitations, BitNet b1.58 [31] refined the approach by introducing a ternary representation (-1, 0, +1) for weights, effectively using 1.58 bits per weight. This modification aimed to balance extreme quantization with model performance, claiming to match full-precision (FP16) LLMs in perplexity and task benchmarks. However, reproducibility remains a challenge, particularly for smaller models [22].
BitNet's performance appears to improve with scale, as demonstrated by successful training on the RedPajama dataset [38, 1], suggesting that this extreme quantization approach may be more suitable for very large models. While BitNet represents a promising direction in pushing the boundaries of model compression, further research is needed to fully understand its applicability across different model sizes and tasks, and to develop reliable training techniques for such extremely quantized models.

3.3 Low-Rank Adaptation (LoRA) and Its Variants

Low-Rank Adaptation (LoRA) [21] is a technique that significantly reduces memory costs during LLM fine-tuning. LoRA achieves this by freezing the original model weights and introducing small, trainable rank decomposition matrices. Specifically, it decomposes the original n×m weight matrix into two smaller matrices of sizes n×r and r×m, where r is a hyperparameter much smaller than m. This approach drastically reduces the number of trainable parameters, leading to substantial memory savings.

3.3.1 GaLORE: Gradient-based Low-Rank Adaptation

GaLORE [49] represents a significant advancement in the LoRA family of techniques by applying low-rank decomposition to gradients rather than weights. This novel approach primarily targets the memory-intensive optimizer states, achieving up to 65.5% reduction in their memory usage. GaLORE is particularly effective for optimizers like Adam and AdamW that heavily rely on component-wise gradient statistics.The core idea of GaLORE is to approximate the full gradient with a low-rank representation. The weight update in GaLORE is formulated as:

$$W_T = W_0 + \sum_{t=0}^{T-1} \tilde{G}_t, \quad \tilde{G}_t = P_t(P_t^T G_t Q_t)Q_t^T$$

where W_T is the weight at step T, W_0 is the initial weight, and $\tilde{G}_t$ is the low-rank approximation of the gradient at step t. P_t and Q_t are low-rank matrices that capture the essential information of the gradient.

Recent developments like Q-GaLORE [47] have further improved efficiency by combining GaLORE with quantization techniques. This hybrid approach has shown impressive results, reducing memory usage from 26 GB VRAM (using 8-bit ADAM) to 18 GB (using 8-bit ADAM with GaLORE). However, GaLORE is not without limitations. The low-rank approximation of gradients may not capture all nuances of the original gradient information, potentially affecting the optimization process in subtle ways. Additionally, the effectiveness of GaLORE can vary depending on the chosen rank, introducing another hyperparameter to tune. The method also adds computational overhead in calculating the low-rank approximations, which may impact training speed, especially for smaller models where memory constraints are less severe.

3.3.2 DoRA: Decomposed Low-Rank Adaptation

DoRA [43] takes a different approach to refining LoRA, focusing on the nature of weight updates rather than their rank. Instead of using an arbitrary rank hyperparameter, DoRA decomposes weight updates into magnitude and direction components. This decomposition aims to capture more nuanced updates, potentially leading to improved accuracy and performance closer to that of full fine-tuning.
The key insight of DoRA is that the importance of a weight update can be separated into how much the weight should change (magnitude) and in what way it should change (direction). By treating these components separately, DoRA allows for more flexible and potentially more effective parameter updates. This approach is particularly beneficial in scenarios where certain weights require significant changes in magnitude but minor adjustments in direction, or vice versa.
While DoRA shows promise in providing more nuanced updates, it takes up more VRAM than other LORA methods, due to the complexity of separating the LORA into magnitude and direction, which might be undesirable when trying to attain maximum compression of LLMs for training.

3.3.3 QLORA: Quantized Low-Rank Adaptation

QLORA [14] represents a significant advancement by combining quantization techniques with LoRA to maximize memory savings in LLMs. At its core, QLORA employs a custom quantization format called Normal Float 4, based on the hypothesis that weights are normally distributed within each matrix. This format optimizes the uniform distribution of weights across all quantization bins. To handle outliers effectively, QLORA introduces k-blockwise quantization, performing quantization separately in smaller blocks across the entire matrix. The method further enhances memory efficiency by implementing a double quantization process: first quantizing the weights, and then quantizing the original quantization constants, such as scaling factors and zero-point offsets. Additionally, QLORA utilizes a paged optimizer, allowing for the offloading of any out-of-memory operations to the CPU. Through these innovative techniques, QLORA achieves remarkable memory savings while maintaining performance comparable to full-precision fine-tuning, representing a significant step forward in the efficient adaptation of large language models.

3.3.4 LORAs for Inference

While Low-Rank Adaptation (LoRA) is primarily known for its efficiency in fine-tuning, it also offers potential benefits for inference. However, the application of LoRA during inference requires a different approach than its use in training. Instead of applying LoRA directly to the model parameters, which would counterintuitively increase memory usage due to the additional LoRA weights, recent research has explored applying LoRA techniques to the Key-Value (KV) cache. One notable method in this direction is PALU [7], which we discuss in more detail in Section 5.1. PALU applies the concept of low-rank decomposition to the KV cache, effectively reducing its memory footprint while maintaining model performance. This approach demonstrates the versatility of LoRA-based techniques in addressing memory constraints across different aspects of LLM operations.

Despite their utility in both fine-tuning and inference, LoRA methods come with their own set of challenges. Biderman et al. [5] highlight several important considerations that underscore the complexity of implementing LoRA effectively. Firstly, LoRA's performance is highly sensitive to hyperparameters, particularly learning rates, making it challenging to achieve optimal results consistently across different models and tasks. Secondly, the effectiveness of LoRA can vary significantly depending on which modules of the model are chosen for adaptation, adding another layer of complexity to its implementation. This inconsistency in performance across different scenarios emphasizes the need for thorough testing and validation when employing LoRA techniques. These challenges highlight the need for careful consideration and empirical validation when implementing LoRA-based methods, whether for fine-tuning or inference optimization. As research in this area progresses, addressing these issues will be crucial for broader and more reliable adoption of LoRA techniques in LLM deployment and optimization. The potential benefits of LoRA in reducing memory requirements and improving efficiency continue to drive interest in overcoming these obstacles, pointing towards a future where LoRA and similar techniques play an increasingly important role in making large language models more accessible and deployable across a wider range of hardware configurations.

3.4 Attention-based Optimization

Attention mechanisms are crucial in LLM architectures but often contribute significantly to memory costs. This section explores techniques that optimize attention to reduce memory usage while maintaining model performance.

3.4.1 Multi-Query Attention (MQA)

Multi-Query Attention [37] simplifies the traditional attention mechanism by using a single key for all queries. While this reduces memory usage and computational costs, it can lead to training instabilities and performance trade-offs due to the reduced expressiveness of the attention mechanism.

3.4.2 Group-Query Attention (GQA)

Group-Query Attention [3] improves upon MQA by allowing multiple queries to share a key, striking a balance between efficiency and performance. This approach reduces the number of parameters and optimizes the key-value cache during inference, leading to memory savings without sacrificing too much model quality.

3.4.3 Flash Attention

Flash Attention [12] takes a different approach by focusing on efficient memory access patterns. It utilizes the GPU's faster but smaller SRAM instead of the larger but slower HBM for attention computations. By breaking down attention calculations into smaller chunks and using an online softmax algorithm, Flash Attention significantly reduces memory usage and improves speed. This method is particularly effective for long sequences, where traditional attention mechanisms struggle with quadratic memory growth.
Subsequent iterations of Flash Attention (versions 2 and 3) [12, 36] have further improved computational efficiency, although their focus has shifted more towards reducing computation time rather than memory usage.

3.4.4 Sliding Window Attention

Longformer's Sliding Window Attention [4] addresses the challenge of processing long sequences by limiting the attention scope of each token to a fixed-size window of surrounding tokens. This approach reduces the computational and memory complexity from quadratic to linear with respect to sequence length. By only attending to a local context, Sliding Window Attention enables the processing of much longer sequences than traditional full attention mechanisms, making it particularly useful for tasks involving extensive documents or continuous streams of text.

These attention optimization techniques demonstrate the ongoing efforts to make LLMs more efficient and scalable. Each method offers a unique trade-off between memory usage, computational speed, and model performance, allowing researchers and practitioners to choose the most suitable approach for their specific needs and use cases.

4 Training-specific Memory Optimization

Training LLMs requires managing multiple memory-intensive components: model parameters, activations, gradients, and optimizer states. This section explores techniques to reduce memory usage during training. Table 3 offers an overview of all training related materials covered in this survey paper.

Table 3: Comparison of Training-Specific Memory Optimization Techniques for LLMs

Method	Papers	Pros	Cons
Activation Checkpointing [10, 6],	Colossal Auto [27]	Significant memory savings Enables training of larger models Compatible with parallel training	Increases computation time Requires careful implementation May impact training speed
Quantized Optimizers	8-bit Optimizers [13]	Reduces optimizer memory usage Enables training of larger models Minimal impact on convergence	Potential impact on training dynamics May require careful tuning Limited by quantization precision
Optimizer State Offloading	ZeRO-Offload [35] Chen et al. [9]	Allows training of very large models Utilizes CPU memory effectively Scalable to distributed systems	Potential slowdown due to data transfer Limited by CPU-GPU bandwidth Increased system complexity

4.1 Activation Checkpointing

Activation checkpointing [6] saves memory by not storing all activations during the forward pass. Instead, it recomputes them during backpropagation. This technique trades some computation time for significant memory savings, making it possible to train larger models on limited hardware. Recent work like Colossal Auto [27] has extended this approach to parallel training setups, further improving efficiency.

4.2 Quantized Optimizers

Optimizers like Adam and AdamW typically store two floating-point states per parameter, consuming substantial memory for large models. Quantized optimizers reduce this memory footprint by using lower-precision representations. For instance, Dettmers et al. [13] demonstrated 8-bit quantization of optimizer states, using techniques like block-wise quantization and dynamic quantization to maintain stability and performance. This is done by using blockwise quantisation, to reduce the number of outliers when doing quantisation by quantising the tensor blockwise instead of the full tensor.

4.3 Optimizer State Offloading

Offloading optimizer states to CPU memory is another effective strategy. Techniques like ZeRO-3 Offload and ZeRO-Infinity [35, 34] move activation memory to the CPU from the GPU to prevent out of memory errors. While this can be limited by CPU-GPU communication bandwidth, recent work by Chen et al. [9] uses learned subspace projectors and efficient communication scheduling

to minimize the performance impact, achieving only a 31% slowdown compared to unlimited GPU memory scenarios.

These techniques, often used in combination, enable the training of increasingly large language models on constrained hardware resources, democratizing access to LLM development and research.

5 Inference Memory Optimization

During inference, the Key-Value (KV) cache becomes a major memory bottleneck, especially for longer sequences. This section explores techniques to optimize memory usage during inference, focusing on KV cache management and related strategies. Table 4 looks at an overview of all inference related methods for memory optimization discussed in this paper.

Table 4: Comparison of Inference-Specific Memory Optimization Techniques for LLMs

Method	Papers	Pros	Cons
KV Cache Quantization	KIVI [29] PALU [7]	Reduces memory footprint of cached vectors Enables longer context handling No retraining required	Potential impact on output quality May require careful implementation Possible speed-memory trade-off
PagedAttention	PagedAttention [25]	Reduces memory fragmentation Improves memory utilization efficiency Handles variable sequence lengths well	Increased implementation complexity May have overhead for short sequences Potential impact on cache efficiency
Token Eviction	H2O [48] NaCL [11]	Maintains small KV cache size Enables processing of very long sequences Adaptive to different text structures	Potential loss of important context Task-dependent effectiveness Complexity in token importance estimation

5.1 KV Cache Quantization

KV cache quantization aims to reduce the memory footprint of cached key and value vectors, which can become a significant bottleneck in long-context inference. This technique, however, must balance memory savings with maintaining model performance.

5.1.1 KIVI

KIVI [29] introduces a tuning-free 2-bit quantization for KV caches, offering a substantial reduction in memory usage. It employs a hybrid approach, using channel-wise quantization for the key cache and token-wise quantization for the value cache. This strategy is based on the observation that keys and values have different sensitivity to quantization errors. Channel-wise quantization for keys helps preserve directional information crucial for attention calculations, while token-wise quantization for values maintains token-specific information.

KIVI's main advantage lies in its simplicity and effectiveness, requiring no additional training or complex tuning processes. However, it faces challenges in maintaining accuracy for extremely long sequences or models with specialized attention mechanisms. The 2-bit quantization, while effective, may sometimes lead to information loss in nuanced language understanding tasks.

5.1.2 PALU

PALU [7] takes a different approach by applying the concept of Low-Rank Adaptation (LoRA) to KV cache compression. It decomposes linear layers into low-rank matrices and caches these smaller intermediate states, reconstructing full keys and values on-the-fly during computation. This method leverages the insight that much of the information in KV caches is redundant or low-rank.

PALU's strength lies in its ability to achieve high compression rates while maintaining model quality. It's particularly effective for models with large hidden sizes. However, the on-the-fly reconstruction introduces additional computational overhead, which can impact inference speed. Balancing this trade-off between memory savings and computational cost is a key challenge for PALU implementation.

5.2 PagedAttention

PagedAttention [25] addresses memory fragmentation issues in KV cache management, a problem that becomes increasingly significant with longer sequences and diverse batch sizes. It partitions the KV cache into blocks, each storing vectors for a pre-allocated number of tokens. This approach:

- Reduces internal fragmentation by avoiding over-allocation of memory.
- Minimizes external fragmentation through non-contiguous memory block storage.
- Improves memory utilization efficiency during inference.

The key innovation of PagedAttention is its ability to manage memory more flexibly, allowing for efficient handling of variable-length sequences and dynamic batch sizes. This is particularly beneficial in production environments where input characteristics

can vary widely.However, PagedAttention faces challenges in optimizing block sizes for diverse model architectures and sequence lengths. Too small blocks can lead to increased management overhead, while too large blocks may reintroduce fragmentation issues. Additionally, the non-contiguous memory access pattern may impact cache efficiency on some hardware architectures, requiring careful implementation to maintain inference speed.

5.3 Token Eviction

Token eviction techniques aim to maintain a small KV cache size while preserving model performance, addressing the challenge of ever-growing memory requirements for long-context processing. The key idea is to selectively retain only the most important tokens in the cache, based on their perceived relevance to the current context.
Notable approaches include:

- H2O [48]: Retains "Heavy Hitter" tokens based on attention score patterns, achieving performance similar to full KV caches with only 20% of the original cache size. This method is effective but can struggle with identifying truly important tokens in complex, long-range dependencies.
- NaCL [11]: Improves upon H2O by evicting tokens in the encoding phase and using proxy tokens unique to each input sequence, enhancing performance on long-context tasks. NaCL addresses some limitations of H2O but introduces additional complexity in the token selection process.

While these token eviction methods show promise in reducing memory requirements, there is still further work that can be done, such as improving the performance of these methods on long context tasks. NaCL takes the first step by introducing a random sampling mechanism on the attention scores to add more robustness, but there is room for future heuristics to improve upon a naive random sampling method.

These inference optimization techniques demonstrate the ongoing efforts to make LLM inference more memory-efficient and scalable on consumer hardware. Each method offers unique trade-offs between memory usage, computational overhead, and model performance.

6 Challenges and Future Directions in Optimizing Memory

As Large Language Models (LLMs) continue to grow in size and complexity, optimizing memory usage has become a critical challenge in both training and inference stages. This section explores the key challenges faced by researchers and practitioners in the field of LLM memory optimization, and outlines potential future directions for addressing these issues. The challenges in memory optimization span across various aspects of LLM development and deployment. In training, we face issues such as the lack of standardized benchmarks, the increasing demands of long-context models, and the complexities introduced by multimodal learning. Inference brings its own set of challenges, including the need to mitigate hallucinations in memory-constrained models and the difficulties in efficiently handling multimodal inputs. Overarching these specific challenges are broader concerns about balancing memory efficiency with model performance and adapting optimization techniques to evolving model architectures. Future directions in this field are diverse and promising. They include developing more sophisticated benchmarking tools, exploring novel attention mechanisms, advancing multimodal optimization techniques, and investigating cutting-edge hardware solutions. The following subsections delve into these challenges and future directions in more detail, providing a comprehensive overview of the landscape of memory optimization in LLMs.

6.1 Training Challenges

6.1.1 Lack of Standardized Benchmarks

The absence of standardized benchmarks hinders direct comparison of memory compression techniques across different hardware configurations, model sizes, and tasks. This challenge impedes progress in the field by making it difficult to accurately assess the relative merits of various optimization approaches.

Future Directions:

- Develop comprehensive benchmark suites that account for various GPU architectures and model parameters.
- Create an automated evaluation harness for testing at different sequence lengths, similar to recent proposals in the field.
- Establish a centralized platform for reporting and comparing memory optimization results, facilitating easier collaboration and comparison among researchers.

6.1.2 Long Context Memory Usage

Training models with extended context windows (e.g., GPT-4's 64k tokens) requires significantly more memory due to the quadratic nature of attention mechanisms. This challenge becomes increasingly critical as models aim to process longer sequences of input data.

Future Directions:

- Refine techniques like Ring Attention [26] to further reduce memory requirements for long-context training.
- Explore novel attention mechanisms that scale linearly or sub-quadratically with sequence length.
- Develop efficient data processing techniques to handle long-context datasets effectively.
- Investigate hardware-specific optimizations for long-sequence processing.

6.1.3 Multimodality in Training

Multimodal models, particularly those using Vision Transformers (ViTs), face increased memory demands due to the tokenization of high-dimensional inputs like images. This challenge is particularly relevant as LLMs expand beyond text to incorporate various types of data.

Future Directions:

- Develop more efficient image tokenization methods that maintain high fidelity while reducing memory footprint.
- Explore adaptive tokenization techniques that adjust based on input complexity and available resources.
- Investigate cross-modal attention mechanisms that optimize memory usage across different modalities.
- Research memory-efficient architectures specifically designed for multimodal learning.

6.2 Inference Challenges

6.2.1 Mitigating Hallucinations

Aggressive memory optimization techniques may lead to increased hallucinations or inaccurate outputs, potentially resulting in harmful or toxic responses. This challenge highlights the delicate balance between memory efficiency and output quality.

Future Directions:

- Develop robust evaluation metrics to assess the impact of memory optimization on output quality and safety.
- Research adaptive compression techniques that dynamically adjust based on the criticality of the task.
- Explore methods to quantify and minimize the trade-off between memory efficiency and output reliability.
- Investigate post-processing techniques to detect and mitigate hallucinations in memory-optimized models.

6.2.2 Multimodality in Inference

Existing KV cache compression methods are primarily optimized for text-based models, leaving a gap for efficient multimodal inference. This challenge becomes increasingly important as LLMs are applied to diverse types of data inputs.

Future Directions:

- Extend KV cache compression techniques to handle diverse data types more efficiently.
- Develop modality-specific caching strategies that account for the unique characteristics of each input type.
- Research unified cache architectures that can efficiently handle multiple modalities simultaneously.
- Investigate dynamic cache allocation methods that adapt to the current multimodal input composition.

6.3 Overarching Challenges and Directions

Beyond the specific challenges in training and inference, there are several overarching issues that span the entire field of LLM memory optimization.

Challenges:

- Balancing the trade-offs between memory efficiency, computational speed, and model performance.
- Ensuring that memory optimization techniques remain effective as model architectures evolve.
- Addressing the increasing complexity of deploying memory-optimized models in production environments.

Future Directions:

- Develop holistic optimization frameworks that jointly consider memory, computation, and performance.
- Explore the potential of neuromorphic computing and other novel hardware architectures for memory-efficient AI.
- Investigate the intersection of memory optimization with other critical areas such as energy efficiency and model interpretability.
- Foster collaboration between hardware manufacturers and AI researchers to co-design optimized solutions.

By addressing these challenges and pursuing these future directions, the field of LLM memory optimization can continue to advance, enabling the development and deployment of more efficient and powerful language models.

7 Conclusion

In conclusion, I have summarized some of the popular methods used nowadays to help reduce memory costs for both inference and training. We also highlighted the importance of these methods in the field by estimating the memory costs on a native finetune, as well as going through more recent papers to construct a better view of the current frontier of memory optimizations in training and inference. As models scale upwards and become more demanding on memory, it is imperative for users without access to datacentre GPUs to innovate and push for memory efficient training and inference methods in order to maximise their benefits from the wave of new LLMs that are becoming more intelligent yet computationally expensive.

References

[1] 1bitLLM. bitnet_b1_58-3b. https://huggingface.co/1bitLLM/bitnet_b1_58-3B, 2024.

[2] Abdin, M., Aneja, J., Awadalla, H., Awadallah, A., Awan, A. A., Bach, N., Bahree, A., Bakhtiari, A., Bao, J., Behl, H., Benhaim, A., Bilenko, M., Bjorck, J., Bubeck, S., Cai, M., Cai, Q., Chaudhary, V., Chen, D., Chen, D., Chen, W., Chen, Y.-C., Chen, Y.-L., Cheng, H., Chopra, P., Dai, X., Dixon, M., Eldan, R., Fragoso, V., Gao, J., Gao, M., Gao, M., Garg, A., Giorno, A. D., Goswami, A., Gunasekar, S., Haider, E., Hao, J., Hewett, R. J., Hu, W., Huynh, J., Iter, D., Jacobs, S. A., Javaheripi, M., Jin, X., Karampatziakis, N., Kauffmann, P., Khademi, M., Kim, D., Kim, Y. J., Kurilenko, L., Lee, J. R., Lee, Y. T., Li, Y., Li, Y., Liang, C., Liden, L., Lin, X., Lin, Z., Liu, C., Liu, L., Liu, M., Liu, W., Liu, X., Luo, C., Madan, P., Mahmoudzadeh, A., Majercak, D., Mazzola, M., Mendes, C. C. T., Mitra, A., Modi, H., Nguyen, A., Norick, B., Patra, B., Perez-Becker, D., Portet, T., Pryzant, R., Qin, H., Radmilac, M., Ren, L., de Rosa, G., Rosset, C., Roy, S., Ruwase, O., Saarikivi, O., Saied, A., Salim, A., Santacroce, M., Shah, S., Shang, N., Sharma, H., Shen, Y., Shukla, S., Song, X., Tanaka, M., Tupini, A., Vaddamanu, P., Wang, C., Wang, G., Wang, L., Wang, S., Wang, X., Wang, Y., Ward, R., Wen, W., Witte, P., Wu, H., Wu, X., Wyatt, M., Xiao, B., Xu, C., Xu, J., Xu, W., Xue, J., Yadav, S., Yang, F., Yang, J., Yang, Y., Yang, Z., Yu, D., Yuan, L., Zhang, C., Zhang, C., Zhang, J., Zhang, L. L., Zhang, Y., Zhang, Y., Zhang, Y., and Zhou, X. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[3] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.

[4] Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *ArXiv abs/2004.05150* (2020).

[5] Biderman, D., Ortiz, J. G., Portes, J., Paul, M., Greengard, P., Jennings, C., King, D., Havens, S., Chiley, V., Frankle, J., Blakeney, C., and Cunningham, J. P. Lora learns less and forgets less. *ArXiv abs/2405.09673* (2024).

[6] Bulatov, Y. Fitting larger networks into memory. https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9, 2018. Accessed on [insert access date here].

[7] Chang, C.-C., Lin, W.-C., Lin, C.-Y., Chen, C.-Y., Hu, Y.-F., Wang, P.-S., Huang, N.-C., Ceze, L., and Wu, K.-C. Palu: Compressing kv-cache with low-rank projection. *ArXiv abs/2407.21118* (2024).

[8] Chavan, A., Magazine, R., Kushwaha, S., Debbah, M., and Gupta, D. Faster and lighter llms: A survey on current challenges and way forward. *ArXiv abs/2402.01799* (2024).

[9] Chen, S., Guan, Z., Liu, Y., and Gibbons, P. B. Practical offloading for fine-tuning llm on commodity gpu via learned subspace projectors, 2024.

[10] Chen, T., Xu, B., Zhang, C., and Guestrin, C. Training deep nets with sublinear memory cost, 2016.

[11] Chen, Y., Wang, G., Shang, J., Cui, S., Zhang, Z., Liu, T., Wang, S., Sun, Y., Yu, D., and Wu, H. Nacl: A general and effective kv cache eviction framework for llm at inference time. *ArXiv abs/2408.03675* (2024).

[12] Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and R'e, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *ArXiv abs/2205.14135* (2022).

[13] Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. 8-bit optimizers via block-wise quantization. *ArXiv abs/2110.02861* (2021).

[14] Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms, 2023.

[15] Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Tan, X. E., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Grattafiori, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Vaughan, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Franco, A.,

Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Wyatt, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Ozgenel, F., Caggioni, F., Guzmán, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Thattai, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Damlaj, I., Molybog, I., Tufanov, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Prasad, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Huang, K., Chawla, K., Lakhotia, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Tsimpoukelli, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Laptev, N. P., Dong, N., Zhang, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Li, R., Hogan, R., Battey, R., Wang, R., Maheswari, R., Howes, R., Rinott, R., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Kohler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Albiero, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wang, X., Wu, X., Wang, X., Xia, X., Wu, X., Gao, X., Chen, Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Hao, Y., Qian, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., and Zhao, Z. The llama 3 herd of models, 2024.

[16] Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *ArXiv abs/2210.17323* (2022).

[17] Gao, C., Chen, J., Zhao, K., Wang, J., and Jing, L. 1-bit fqt: Pushing the limit of fully quantized training to 1-bit, 2024.

[18] Gong, R., Ding, Y., Wang, Z., Lv, C., Zheng, X., Du, J., Qin, H., Guo, J., Magno, M., and Liu, X. A survey of low-bit large language models: Basics, systems, and algorithms.

[19] Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces, 2022.

[20] Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J. W., Vinyals, O., and Sifre, L. Training compute-optimal large language models, 2022.

[21] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models, 2021.

[22] joey00072. Experiments with bitnet-1.5, 2024.

[23] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models, 2020.

[24] Kim, J., Lee, J. H., Kim, S., Park, J., Yoo, K. M., Kwon, S. J., and Lee, D. Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization, 2023.

[25] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention, 2023.

[26] Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *ArXiv abs/2310.01889* (2023).

[27] Liu, Y., Li, S., Fang, J., Shao, Y., Yao, B., and You, Y. Colossal-auto: Unified automation of parallelization and activation checkpoint for large-scale models. *ArXiv abs/2302.02599* (2023).

[28] Liu, Z., Oguz, B., Zhao, C., Chang, E., Stock, P., Mehdad, Y., Shi, Y., Krishnamoorthi, R., and Chandra, V. LLM-QAT: Data-free quantization aware training for large language models, 2024.

[29] Liu, Z., Yuan, J., Jin, H., Zhong, S., Xu, Z., Braverman, V., Chen, B., and Hu, X. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. In *Forty-first International Conference on Machine Learning* (2024).

[30] Micikevicius, P., Narang, S., Alben, J., Diamos, G. F., Elsen, E., García, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training. *CoRR abs/1710.03740* (2017).

[31] Nielsen, J., and Schneider-Kamp, P. Bitnet b1.58 reloaded: State-of-the-art performance also on smaller networks. *ArXiv abs/2407.09527* (2024).

[32] NVIDIA. Mastering LLM techniques: Inference optimization. `https://developer.nvidia.com/blog/mastering-llm-techniques-inference-optimization/`, 2023. Accessed on [insert access date here].

[33] Peng, B., Alcaide, E., Anthony, Q., Albalak, A., Arcadinho, S., Biderman, S., Cao, H., Cheng, X., Chung, M., Grella, M., GV, K. K., He, X., Hou, H., Lin, J., Kazienko, P., Kocon, J., Kong, J., Koptyra, B., Lau, H., Mantri, K. S. I., Mom, F., Saito, A.,

Song, G., Tang, X., Wang, B., Wind, J. S., Wozniak, S., Zhang, R., Zhang, Z., Zhao, Q., Zhou, P., Zhou, Q., Zhu, J., and Zhu, R.-J. Rwkv: Reinventing rnns for the transformer era, 2023.

[34] Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis* (2021), 1–15.

[35] Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., Li, D., and He, Y. Zero-offload: Democratizing billion-scale model training. *ArXiv abs/2101.06840* (2021).

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

[37] Shazeer, N. M. Fast transformer decoding: One write-head is all you need. *ArXiv abs/1911.02150* (2019).

[38] Together AI. Redpajama-data. `https://github.com/togethercomputer/RedPajama-Data`, 2023.

[39] turboderp. Exllamav2. `https://github.com/turboderp/exllamav2`, 2024. Accessed on 11/10/2024.

[40] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

[41] Wang, H., Ma, S., Dong, L., Huang, S., Wang, H., Ma, L., Yang, F., Wang, R., Wu, Y., and Wei, F. Bitnet: Scaling 1-bit transformers for large language models. *ArXiv abs/2310.11453* (2023).

[42] Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Tang, J., Wang, J., Yang, J., Tu, J., Zhang, J., Ma, J., Xu, J., Zhou, J., Bai, J., He, J., Lin, J., Dang, K., Lu, K., Chen, K.-Y., Yang, K., Li, M., Xue, M., Ni, N., Zhang, P., Wang, P., Peng, R., Men, R., Gao, R., Lin, R., Wang, S., Bai, S., Tan, S., Zhu, T., Li, T., Liu, T., Ge, W., Deng, X., Zhou, X., Ren, X., Zhang, X., Wei, X., Ren, X., Fan, Y., Yao, Y., Zhang, Y., Wan, Y., Chu, Y., Cui, Z., Zhang, Z., and Fan, Z.-W. Qwen2 technical report. *ArXiv abs/2407.10671* (2024).

[43] yang Liu, S., Wang, C.-Y., Yin, H., Molchanov, P., Wang, Y.-C. F., Cheng, K.-T., and Chen, M.-H. Dora: Weight-decomposed low-rank adaptation. *ArXiv abs/2402.09353* (2024).

[44] Ye, T., Dong, L., Xia, Y., Sun, Y., Zhu, Y., Huang, G., and Wei, F. Differential transformer, 2024.

[45] Zhang, T., Yi, J., Xu, Z., and Shrivastava, A. Kv cache is 1 bit per channel: Efficient large language model inference with coupled quantization, 2024.

[46] Zhang, Y., Zhang, P., Huang, M., Xiang, J., Wang, Y., Wang, C., Zhang, Y., Yu, L., Liu, C., and Lin, W. Qqq: Quality quattuor-bit quantization for large language models. *ArXiv abs/2406.09904* (2024).

[47] Zhang, Z., Jaiswal, A., Yin, L., Liu, S., Zhao, J., Tian, Y., and Wang, Z. Q-galore: Quantized galore with int4 projection and layer-adaptive low-rank gradients, 2024.

[48] Zhang, Z. A., Sheng, Y., Zhou, T., Chen, T., Zheng, L., Cai, R., Song, Z., Tian, Y., Ré, C., Barrett, C. W., Wang, Z., and Chen, B. H2o: Heavy-hitter oracle for efficient generative inference of large language models. *ArXiv abs/2306.14048* (2023).

[49] Zhao, J., Zhang, Z. A., Chen, B., Wang, Z., Anandkumar, A., and Tian, Y. Galore: Memory-efficient llm training by gradient low-rank projection. *ArXiv abs/2403.03507* (2024).

[50] Zhou, Z., Ning, X., Hong, K., Fu, T., Xu, J., Li, S., Lou, Y., Wang, L., Yuan, Z., Li, X., Yan, S., Dai, G., Zhang, X.-P., Dong, Y., and Wang, Y. A survey on efficient inference for large language models. *ArXiv abs/2404.14294* (2024).

8 Appendix

8.1 Pruning and Sparse Networks

Back in 2016, Frankle et. al published the Lottery Ticket Hypothesis, which proved that every neural network had a sparser version that could achieve the same result. This led to many studies such as pruning and the creation of sparse networks to try to maximise performance whilst minimizing memory usage and compute.

8.1.1 Pruning

Pruning is the act of removing unimportant network parameters from the LLM, in order to preserve performance whilst improving speed and memory. An early study of pruning LLMs came from LLM Pruner, which pruned LLMs by reducing the number of attention heads and their associated weight connections in the model, while maintaining the number of layers in the LLM. This solved issues with other pruning methods such as xxx and yyy, which had to further do post training on the smaller model, taking up more time. Moreover, future work improved on reducing the size of the pruned model further by quantising the model or applying LORA on it. Additionally, there has also been some research on other types of pruning,such as block pruning which prunes a model block by block.

8.2 Alternative LLM architectures

8.2.1 RWKV

RWKV is an alternative to the attention layer. ooLater versions such as Eagle, Finch also use the same structure, but use slight improvemenmts such as linear interpolation in order to imrpove the model's performance.

8.2.2 State Space Models

S4 H3 Mamba