Efficient Deployment Algorithms for Large Language Models

by

Guangxuan Xiao

B.Eng., Tsinghua University (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Guangxuan Xiao Department of Electrical Engineering and Computer Science May 17, 2024
Certified by:	Song Han Associate Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Leslie A. Kolodziejski Professor of Electrical Engineering and Computer Science Chair, Department Committee on Graduate Students

Efficient Deployment Algorithms for Large Language Models

by

Guangxuan Xiao

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Large language models (LLMs) have achieved impressive performance on various natural language tasks. However, their massive computational and memory requirements hinder widespread deployment. Additionally, deploying them on extensive inputs presents efficiency and accuracy challenges. This proposal introduces two techniques to enable efficient and accurate quantization and streaming deployment of LLMs, facilitating their application in real-world systems with limited resources. First, we develop SmoothQuant, an accurate post-training 8-bit quantization method of both weights and activations in LLMs up to 530B parameters. By smoothing outliers in activations, SmoothQuant enables the use of efficient INT8 kernels on all matrix multiplications with negligible accuracy loss. Second, we present StreamingLLM, enabling LLMs to handle arbitrarily long text sequences using a fixed memory budget. It exploits "attention sinks" in LLMs to stably anchor attention computation on lengthy contexts. Experiments show StreamingLLM can model over 4 million tokens with up to 22x speedup compared to recomputation baselines. Together, these two techniques can significantly reduce the computational and memory costs of large language models, increasing their accessibility for practical usage.

Thesis supervisor: Song Han
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

I am deeply thankful to my advisor, Professor Song Han, for his exceptional guidance, wisdom, and patience throughout my research journey. His profound knowledge, insightful critiques, and unwavering support have been crucial in shaping my path as a researcher. I am immensely grateful for his encouragement and trust, his dedication to pushing the boundaries of science, and his unrelenting pursuit of making the world a better place through research. Having him as my mentor has been one of the greatest privileges of my academic career.

I also extend my heartfelt thanks to my undergraduate advisors, Professors Zhiyuan Liu and Jidong Zhai, who introduced me to the intriguing worlds of natural language processing, machine learning, and computer systems. Their guidance during my formative years at university has been invaluable in my development as a researcher. Furthermore, I owe a great deal of gratitude to my undergraduate research supervisors, Professors Jiajun Wu, Jure Leskovec, and Leslie Kaelbling, who not only inspired me to pursue a career in academia but also supported me immensely through the PhD application process. Their mentorship has been pivotal in my growth both as a researcher and as a person.

At MIT, I have been fortunate to be taught and mentored by several of the finest minds in their respective fields. My academic advisor, Professor Hari Balakrishnan, has provided constant support and invaluable advice on research direction and career development. I am also thankful to Professors Bill Freeman, Yoon Kim, Jacob Andreas, Luca Daniel, Vivienne Sze, and Joel Emer for their extensive knowledge and mentorship in their areas of expertise including Computer Vision, Natural Language Processing, modeling and simulation methods, and the intricate world of hardware and architecture.

The camaraderie and collaboration in my lab at MIT have greatly enriched my PhD experience. I sincerely thank Dr. Ji Lin, Dr. Zhijian Liu, Dr. Hanrui Wang, Dr. Han Cai, and all my collaborators for their insights and contributions to our joint research endeavors. My gratitude also extends to my labmates Yujun Lin, Haotian Tang, Ligeng Zhu, Zhekai Zhang, Muyang Li, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Jiaming Tang, Xingyu Dang, and Nicole Stiles, whose companionship and support have been a constant source of joy and inspiration. Our discussions, both academic and leisure, and our shared meals and sports activities have created lasting memories.

During my summer internship at Meta AI in 2023, I had the privilege of working under the mentorship of Dr. Mike Lewis, Dr. Yuandong Tian, and Dr. Beidi Chen. This experience not only helped me overcome numerous research challenges but also led to one of my proudest papers, StreamingLLM. I am thankful for their guidance and the collaborative spirit during my internship.

I would also like to express my gratitude to my good friends, Tianwei Yin, Jianke

Yang, Minghao Guo, Tianyuan Zhang, Bowen Pan, Tiancheng Yu, and Haike Xu, for their unwavering support, encouragement, and companionship throughout my time at MIT. Their presence has greatly enriched my experience, making my journey not only more enjoyable but also filled with lasting memories and profound camaraderie. I am immensely thankful for the moments we shared, both in times of challenge and during moments of leisure.

My deepest appreciation goes to my parents, Hongyu Xue and Yong Xiao, for their unconditional love and support. Their faith in my abilities has never wavered, and their encouragement has been a fundamental driving force in my pursuit of academic and personal excellence. Additionally, I am grateful to my cousin Ying Gan and her husband Jian Xu, who have provided a warm familial environment during my time in the United States. Their support through all the ups and downs has been a great comfort and has helped me navigate the challenges of living far from my home country.

This acknowledgment reflects my immense gratitude to each individual and group who has contributed to my journey. I am blessed to have such a supportive and inspiring circle of family, friends, and mentors. Thank you all for your support, encouragement, and love. This thesis is dedicated to all of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large language models (LLMs) are becoming increasingly essential across a variety of natural language processing applications such as dialog systems, document summarization, code completion, and question answering [1]–[14]. However, deploying these models is costly and energy-consuming due to their massive size. For example, the GPT-3 [15] model contains 175 billion parameters, which will consume at least 350GB of memory to store and run in FP16, requiring 8×48GB A6000 GPUs or 5×80GB A100 GPUs just for inference. Due to the huge computation and communication overhead, the inference latency may also be unacceptable to real-world applications.

1.1 Quantization Challenges in LLMs

Quantization is a promising way to reduce the cost of LLMs [16], [17]. By quantizing the *weights and activations* with low-bit integers, we can reduce GPU memory requirements, in size and bandwidth, and accelerate compute-intensive operations (i.e., `GEMM`[*] in linear layers, `BMM`[†] in attention). For instance, INT8 quantization of weights and activations can halve the GPU memory usage and nearly double the throughput of matrix multiplications compared to FP16.

However, unlike CNN models or smaller transformer models like BERT [18], the *activations* of LLMs are difficult to quantize. When we scale up LLMs beyond 6.7B parameters, systematic outliers with large magnitude will emerge in activations [16], leading to large quantization errors and accuracy degradation. ZeroQuant [17] applies dynamic per-token activation quantization and group-wise weight quantization (defined in Figure 3.1 Sec. 3.1). It can be implemented efficiently and delivers good accuracy for GPT-3-350M and GPT-J-6B. However, it cannot maintain the accuracy for the large OPT model with 175 billion parameters (see Section 3.4.2).

`LLM.int8()` [16] addresses that accuracy issue by further introducing a mixed-precision decomposition (i.e., it keeps outliers in FP16 and uses INT8 for the other activations). However, it is hard to implement the decomposition efficiently on hardware accelerators. Therefore, deriving an *efficient*, *hardware-friendly*, and preferably *training-free* quantization

[*]General matrix multiply

[†]Batch matrix multiply

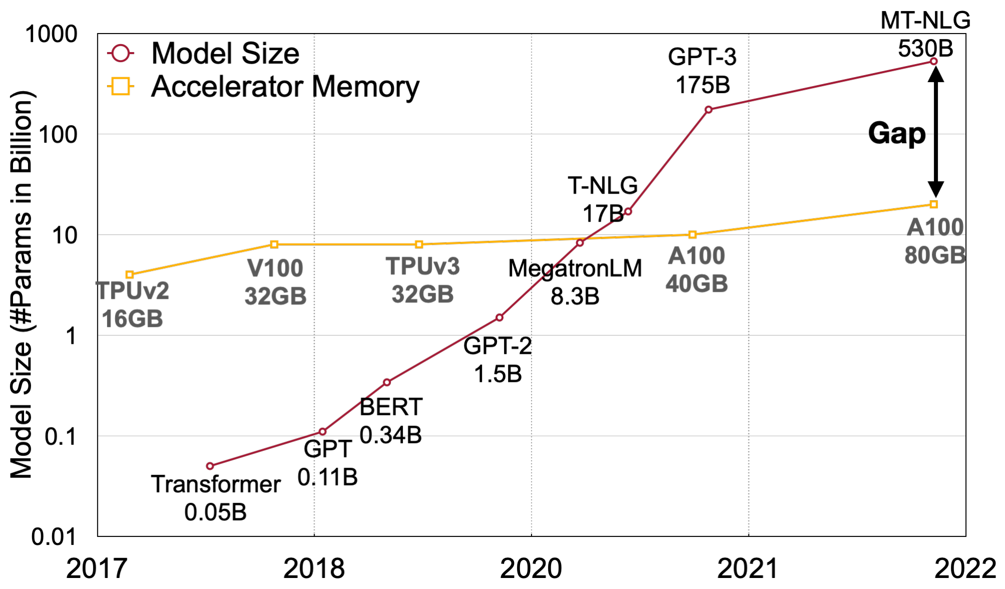


Figure 1.1. The model size of large language models is developing at a faster pace than the GPU memory in recent years, leading to a big gap between the supply and demand for memory. Quantization and model compression techniques can help bridge the gap.

scheme for LLMs that would use INT8 for all the compute-intensive operations remains an open challenge.

We propose SmoothQuant, an accurate and efficient post-training quantization (PTQ) solution for LLMs. SmoothQuant relies on a key observation: even if activations are much harder to quantize than weights due to the presence of outliers [16], different tokens exhibit similar variations across their channels.

Based on this observation, SmoothQuant offline migrates the quantization difficulty from activations to weights (Figure 1.2). SmoothQuant proposes a mathematically equivalent per-channel scaling transformation that significantly smooths the magnitude across the channels, making the model quantization-friendly. Since SmoothQuant is compatible with various quantization schemes, we implement three efficiency levels of quantization settings for SmoothQuant (see Table 3.2, O1-O3).

Experiments show that SmoothQuant is hardware-efficient: it can maintain the performance of OPT-175B [19], BLOOM-176B [20], GLM-130B [21], and MT-NLG 530B [22], leading to up to 1.51× speed up and 1.96× memory saving on PyTorch. SmoothQuant is easy to implement. We integrate SmoothQuant into FasterTransformer, the state-of-the-art transformer serving framework, achieving up to 1.56× speedup and halving the memory usage compared with FP16. Remarkably, SmoothQuant allows serving large models like OPT-175B using only half the number of GPUs compared to FP16 while being faster and enabling the serving of a 530B model within one 8-GPU node. Our work democratizes the use of LLMs by offering a turnkey solution to reduce the serving cost.

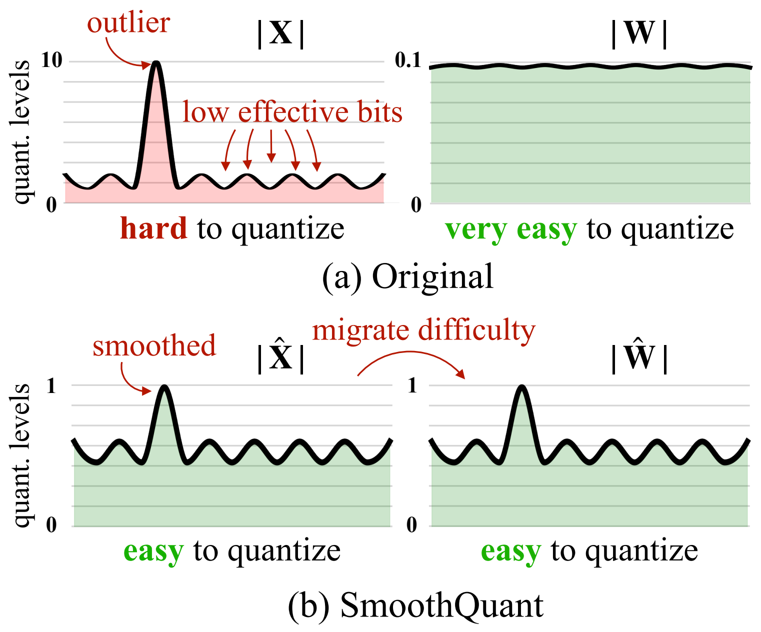


Figure 1.2. SmoothQuant's intuition: the activation $\mathbf{X}$ is hard to quantize because outliers stretch the quantization range, leaving few effective bits for most values. We migrate the scale variance from activations to weights $\mathbf{W}$ during offline to reduce the quantization difficulty of activations. The smoothed activation $\hat{\mathbf{X}}$ and the adjusted weight $\hat{\mathbf{W}}$ are both easy to quantize.

1.2 StreamingLLM for Infinite-Length Inputs

Large Language Models (LLMs) [1]–[6] are becoming ubiquitous, powering many natural language processing applications such as dialog systems [7]–[9], document summarization [10], [11], code completion [12], [13] and question answering [14]. To unleash the full potential of pretrained LLMs, they should be able to efficiently and accurately perform long sequence generation. For example, an ideal ChatBot assistant can stably work over the content of recent day-long conversations. However, it is very challenging for LLMs to generalize to longer sequence lengths than they have been pretrained on, e.g., 4K for Llama-2 [6].

The reason is that LLMs are constrained by the attention window during pre-training. Despite substantial efforts to expand this window size [23]–[25] and improve training [26], [27] and inference [28]–[32] efficiency for lengthy inputs, the acceptable sequence length remains intrinsically *finite*, which doesn't allow persistent deployments.

In this thesis, we first introduce the concept of LLM streaming applications and ask the question:

Can we deploy an LLM for infinite-length inputs without sacrificing efficiency and performance?

When applying LLMs for infinite input streams, two primary challenges arise:

1. During the decoding stage, Transformer-based LLMs cache the Key and Value states (KV) of all previous tokens, as illustrated in Figure 1.3 (a), which can lead to excessive memory usage and increasing decoding latency [28].

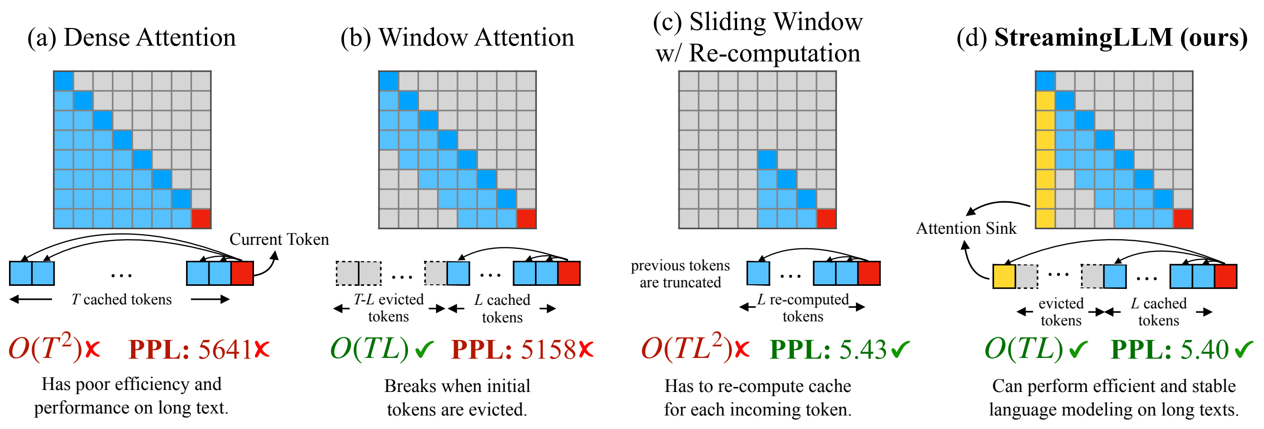


Figure 1.3. Illustration of StreamingLLM *vs.* existing methods. The language model, pre-trained on texts of length L, predicts the Tth token ($T \gg L$). (a) Dense Attention has $O(T^2)$ time complexity and an increasing cache size. Its performance decreases when the text length exceeds the pre-training text length. (b) Window Attention caches the most recent L tokens' KV. While efficient in inference, performance declines sharply once the starting tokens' keys and values are evicted. (c) Sliding Window with Re-computation rebuilds the KV states from the L recent tokens for each new token. While it performs well on long texts, its $O(TL^2)$ complexity, stemming from quadratic attention in context re-computation, makes it considerably slow. (d) StreamingLLM keeps the *attention sink* (several initial tokens) for stable attention computation, combined with the recent tokens. It's efficient and offers stable performance on extended texts. Perplexities are measured using the Llama-2-13B model on the first book (65K tokens) in the PG-19 test set.

2. Existing models have limited length extrapolation abilities, i.e., their performance degrades [23], [33] when the sequence length goes beyond the attention window size set during pre-training.

An intuitive approach, known as window attention [34] (Figure 1.3 b), maintains only a fixed-size sliding window on the KV states of most recent tokens. Although it ensures constant memory usage and decoding speed after the cache is initially filled, the model collapses once the sequence length exceeds the cache size, i.e., *even just evicting the KV of the first token*, as illustrated in Figure 4.1. Another strategy is the sliding window with re-computation (shown in Figure 1.3 c), which rebuilds the KV states of recent tokens for each generated token. While it offers strong performance, this approach is significantly slower due to the computation of quadratic attention within its window, making this method impractical for real-world streaming applications.

To understand the failure of window attention, we find an interesting phenomenon of autoregressive LLMs: a surprisingly large amount of attention score is allocated to the initial tokens, irrespective of their relevance to the language modeling task, as visualized in Figure 1.4. We term these tokens "**attention sinks**". Despite their lack of semantic significance, they collect significant attention scores. We attribute the reason to the Softmax operation, which requires attention scores to sum up to one for all contextual tokens. Thus, even when the current query does not have a strong match in many previous tokens, the model still needs to allocate these unneeded attention values somewhere so it sums up to one. The reason behind *initial* tokens as sink tokens is intuitive: initial tokens are visible to

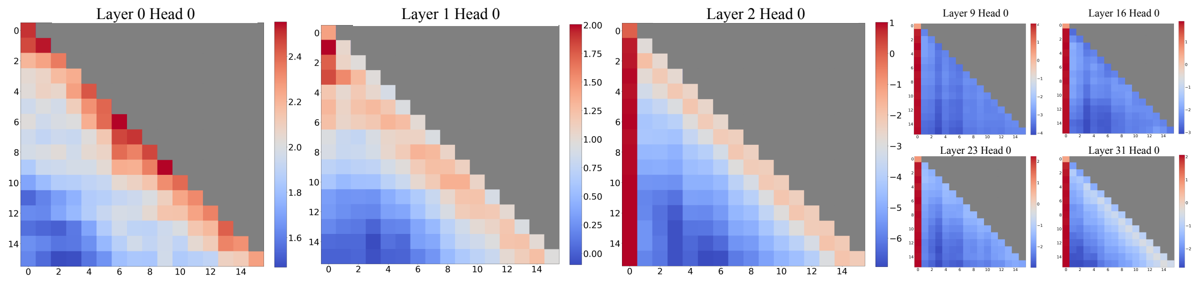


Figure 1.4. Visualization of the *average* attention logits in Llama-2-7B over 256 sentences, each with a length of 16. Observations include: (1) The attention maps in the first two layers (layers 0 and 1) exhibit the "local" pattern, with recent tokens receiving more attention. (2) Beyond the bottom two layers, the model heavily attends to the initial token across all layers and heads.

almost all subsequent tokens because of the autoregressive language modeling nature, making them more readily trained to serve as attention sinks.

Based on the above insights, we propose StreamingLLM, a simple and efficient framework that enables LLMs trained with a finite attention window to work on text of infinite length without fine-tuning. StreamingLLM exploits the fact that attention sinks have high attention values, and preserving them can maintain the attention score distribution close to normal. Therefore, StreamingLLM simply keeps the attention sink tokens' KV (with just 4 initial tokens sufficing) together with the sliding window's KV to anchor the attention computation and stabilize the model's performance. With StreamingLLM, models including Llama-2-[7, 13, 70]B, MPT-[7, 30]B, Falcon-[7, 40]B, and Pythia-[2.9,6.9,12]B can reliably model 4 million tokens, and potentially even more. Compared with the only viable baseline, sliding window with recomputation, StreamingLLM achieves up to 22.2× speedup, realizing the streaming use of LLMs.

Furthermore, we confirm our attention sink hypothesis and demonstrate that language models can be pre-trained to require only a single attention sink token for streaming deployment. Specifically, we suggest that an extra learnable token at the beginning of all training samples can serve as a designated attention sink. By pre-training 160 million parameter language models from scratch, we demonstrate that adding this single sink token preserves the model's performance in streaming cases. This stands in contrast to vanilla models, which necessitate the reintroduction of multiple initial tokens as attention sinks to achieve the same performance level.

Finally, we emphasize that StreamingLLM efficiently generates coherent text from tokens within the KV cache without extending the LLMs' context length. It suits continuous operation needs with minimal memory use and past data reliance. Additionally, StreamingLLM can complement context extension methods to increase the attendable recent context.

Together, SmoothQuant and StreamingLLM are complementary strategies aimed at democratizing the use of LLMs by reducing deployment costs and enabling continuous, real-time applications with potentially infinite inputs. These innovations pave the way for broader and more effective deployment of LLM technologies in diverse real-world scenarios.

Chapter 2

Background and Related Work

2.1 Large Language Models

Pre-trained language models have achieved remarkable performance on various benchmarks by *scaling up*. GPT-3 [2] is the first LLM beyond 100B parameters and achieves impressive few-shot/zero-shot learning results. Later works [22], [35]–[37] continue to push the frontier of scaling, going beyond 500B parameters. However, as the language model gets larger, serving such models for inference becomes expensive and challenging. In this work, we show that our proposed method can quantize the three largest, openly available LLMs: OPT-175B [19], BLOOM-176B [20], and GLM-130B [21], and even MT-NLG 530B [22] to reduce the memory cost and accelerate inference.

2.2 Model Quantization

Quantization is an effective method for reducing the model size and accelerating inference. It proves to be effective for various convolutional neural networks (CNNs) [38]–[42] and transformers [43]–[47]. Weight equalization [40] and channel splitting [48] reduce quantization error by suppressing the outliers in weights. However, these techniques cannot address the activation outliers, which are the major quantization bottleneck for LLMs [16].

2.3 Quantization of Large Language Models

GPTQ [49] applies quantization only to weights but not activations. ZeroQuant [17] and nuQmm [50] use a per-token and group-wise quantization scheme for LLMs, which requires customized CUDA kernels. Their largest evaluated models are 20B and 2.7B, respectively, and fail to maintain the performance of LLMs like OPT-175B. `LLM.int8()` [16] uses mixed INT8/FP16 decomposition to address the activation outliers. However, such implementation leads to large latency overhead, which can be even slower than FP16 inference. Outlier Suppression [51] uses the non-scaling LayerNorm and token-wise clipping to deal with the activation outliers. However, it only succeeds on small language models such as BERT [18] and BART [52] and fails to maintain the accuracy for LLMs (Table 3.4). Our algorithm preserves the performance of LLMs (up to 176B, the largest open-source LLM we can find) with an efficient per-tensor, static quantization scheme without retraining, allowing us to use off-the-shelf INT8 `GEMM` to achieve high hardware efficiency.

2.4 Applying LLMs to Lengthy Texts

Extensive research has been done on applying LLMs to lengthy texts, with three main areas of focus: **Length Extrapolation**, **Context Window Extension**, and **Improving LLMs' Utilization of Long Text**. While seemingly related, it's worth noting that progress in one direction doesn't necessarily lead to progress in the other. For example, extending the context size of LLMs doesn't improve the model's performance beyond the context size, and neither approach ensures effective use of the long context. Our StreamingLLM framework primarily lies in the first category, where LLMs are applied to text significantly exceeding the pre-training window size, potentially even of infinite length. We do not expand the attention window size of LLMs or enhance the model's memory and usage on long texts. The last two categories are orthogonal to our focus and could be integrated with our techniques.

2.4.1 Length Extrapolation

Length extrapolation aims to enable language models trained on shorter texts to handle longer ones during testing. A predominant avenue of research targets the development of relative position encoding methods for Transformer models, enabling them to function beyond their training window. One such initiative is Rotary Position Embeddings (RoPE) [53], which transforms the queries and keys in every attention layer for relative position integration. Despite its promise, subsequent research [23], [33] indicated its underperformance on text that exceeds the training window. Another approach, ALiBi [33], biases the query-key attention scores based on their distance, thereby introducing relative positional information. While this exhibited improved extrapolation, our tests on MPT models highlighted a breakdown when the text length was vastly greater than the training length. Current methodologies, however, have yet to achieve infinite length extrapolation, causing no existing LLMs to fit for streaming applications.

2.4.2 Context Window Extension

Context Window Extension centers on expanding the LLMs' context window, enabling the processing of more tokens in one forward pass. A primary line of work addresses the training efficiency problem. Given the attention to computation's quadratic complexity during training, developing a long-context LLM is both a computational and memory challenge. Solutions have ranged from system-focused optimizations like FlashAttention [26], [27], which accelerates attention computation and reduces memory footprint, to approximate attention methods [34], [54]–[56] that trade model quality for efficiency. Recently, there has been a surge of work on extending pre-trained LLMs with RoPE [23]–[25], [57], involving position interpolation and fine-tuning. However, all the aforementioned techniques only extend LLMs' context window to a limited extent, which falls short of StreamingLLM's primary concern of handling limitless inputs.

2.4.3 Improving LLMs' Utilization of Long Text

Improving LLMs' Utilization of Long Text optimizes LLMs to better capture and employ the content within the context rather than merely taking them as inputs. As highlighted by [58] and [59], success in the previously mentioned two directions does not necessarily translate to competent utilization of lengthy contexts. Addressing this effective usage of prolonged contexts within LLMs is still a challenge. Our work concentrates on stably harnessing the most recent tokens, enabling the seamless streaming application of LLMs.

Chapter 3

SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models

3.1 Preliminaries of Quantization in Neural Networks

Quantization maps a high-precision value into discrete levels. We study integer uniform quantization [39] (specifically INT8) for better hardware support and efficiency. The quantization process can be expressed as:

$$\bar{\mathbf{X}}^{\text{INT8}} = \lceil \frac{\mathbf{X}^{\text{FP16}}}{\Delta} \rfloor, \quad \Delta = \frac{\max(|\mathbf{X}|)}{2^{N-1} - 1}, \tag{3.1}$$

where $\mathbf{X}$ is the floating-point tensor, $\bar{\mathbf{X}}$ is the quantized counterpart, Δ is the quantization step size, $\lceil \cdot \rfloor$ is the rounding function, and N is the number of bits (8 in our case). Here we assume the tensor is *symmetric* at 0 for simplicity; the discussion is similar for asymmetric cases (e.g., after ReLU) by adding a zero-point [39].

Such quantizer uses the maximum absolute value to calculate Δ so that it preserves the outliers in activation, which are found to be important for accuracy [16]. We can calculate Δ offline with the activations of some calibration samples, what we call **static quantization**. We can also use the runtime statistics of activations to get Δ, what we call **dynamic quantization**. As shown in Figure 3.1, quantization has different granularity levels. The **per-tensor** quantization uses a single step size for the entire matrix. We can further enable finer-grained quantization by using different quantization step sizes for activations associated with each token (**per-token** quantization) or each output channel of weights (**per-channel** quantization). A coarse-grained version of per-channel quantization is to use different quantization steps for different channel groups, called **group-wise** quantization [17], [43].

For a linear layer in Transformers [60] $\mathbf{Y} = \mathbf{X} \cdot \mathbf{W}, \mathbf{Y} \in \mathbb{R}^{T \times C_o}, \mathbf{X} \in \mathbb{R}^{T \times C_i}, \mathbf{W} \in \mathbb{R}^{C_i \times C_o}$, where T is the number of tokens, C_i is the input channel, and C_o is the output channel (see Figure 3.1, we omit the batch dimension for simplicity), we can reduce the storage by half compared to FP16 by quantizing the weights to INT8. However, to speed up the inference,

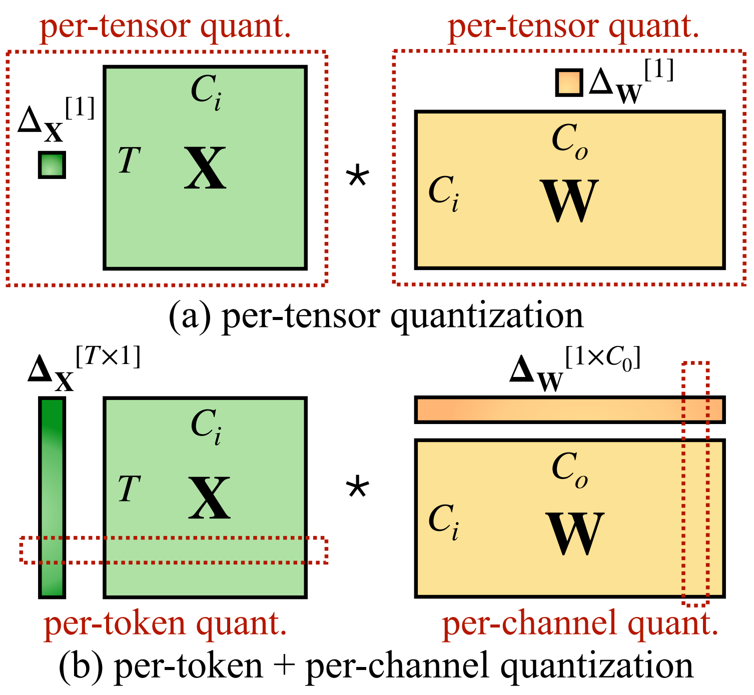

per-tensor quant.
per-tensor quant.
$\Delta_{\mathbf{X}}^{[1]}$
$\Delta_{\mathbf{W}}^{[1]}$
C_i
T
$\mathbf{X}$
$*$
C_o
C_i
$\mathbf{W}$
(a) per-tensor quantization
$\Delta_{\mathbf{X}}^{[T\times 1]}$
$\Delta_{\mathbf{W}}^{[1\times C_0]}$
per-token quant.
per-channel quant.
(b) per-token + per-channel quantization

Figure 3.1. Definition of per-tensor, per-token, and per-channel quantization. Per-tensor quantization is the most efficient to implement. For vector-wise quantization to efficiently utilize the INT8 `GEMM` kernels, we can only use scaling factors from the outer dimensions (i.e., token dimension T and out channel dimension C_o) but not inner dimension (i.e., in channel dimension C_i).

we need to quantize both weights and activations into INT8 (i.e., W8A8) to utilize the integer kernels (e.g., INT8 `GEMM`), which are supported by a wide range of hardware (e.g., NVIDIA GPUs, Intel CPUs, Qualcomm DSPs, etc.).

3.2 Review of Quantization Difficulty

LLMs are notoriously difficult to quantize due to the outliers in the activations [16], [47], [51]. We first review the difficulties of activation quantization and look for a pattern amongst outliers. We visualize the input activations and the weights of a linear layer that has a large quantization error in Figure 3.2 (left). We can find several patterns that motivate our method:

1. Activations are harder to quantize than weights. The weight distribution is quite uniform and flat, which is easy to quantize. Previous work has shown that quantizing the weights of LLMs with INT8 or even with INT4 does not degrade accuracy [16], [17], [21], which echoes our observation.

2. Outliers make activation quantization difficult. The scale of outliers in activations is $\sim 100\times$ larger than most of the activation values. In the case of per-tensor quantization (Equation 3.1), the large outliers dominate the maximum magnitude measurement, leading to low *effective quantization bits/levels* (Figure 1.2) for non-outlier channels: suppose the maximum magnitude of channel i is m_i, and the maximum value of the whole matrix is

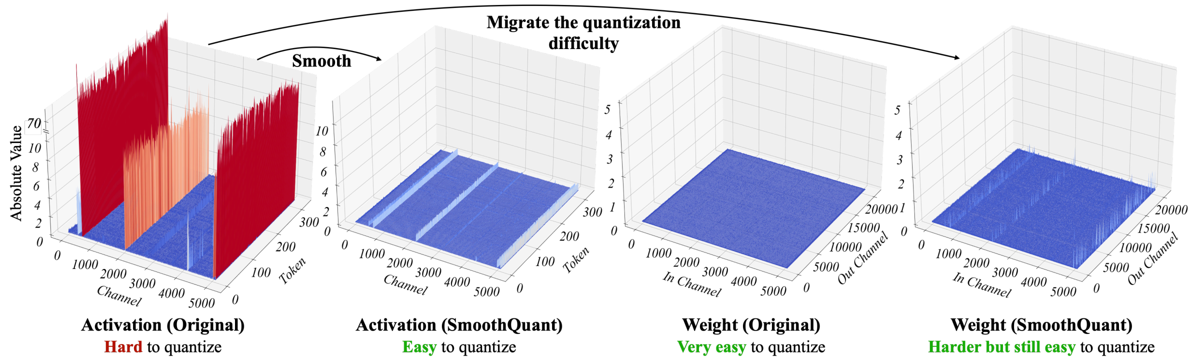


Figure 3.2. Magnitude of the input activations and weights of a linear layer in OPT-13B before and after SmoothQuant. Observations: (1) there are a few channels in the original activation map whose magnitudes are very large (greater than 70); (2) the variance in one activation channel is small; (3) the original weight distribution is flat and uniform. SmoothQuant migrates the outlier channels from activation to weight. In the end, the outliers in the activation are greatly smoothed while the weight is still pretty smooth and flat.

Table 3.1. Among different activation quantization schemes, only per-channel quantization [47] preserves the accuracy, but it is *not* compatible (marked in gray) with INT8 `GEMM` kernels. We report the average accuracy on WinoGrande, HellaSwag, PIQA, and LAMBADA.

Model size (OPT-)	6.7B	13B	30B	66B	175B
FP16	64.9%	65.6%	67.9%	69.5%	71.6%
INT8 per-tensor	39.9%	33.0%	32.8%	33.1%	32.3%
INT8 per-token	42.5%	33.0%	33.1%	32.9%	31.7%
INT8 per-channel	64.8%	65.6%	68.0%	69.4%	71.4%

m, the effective quantization levels of channel i is $2^8 \cdot m_i/m$. For non-outlier channels, the effective quantization levels would be very small (2-3), leading to large quantization errors.

3. Outliers persist in fixed channels. Outliers appear in a small fraction of the *channels*. If one channel has an outlier, it persistently appears in all tokens (Figure 3.2, red). The variance amongst the channels for a given token is large (the activations in some channels are very large, but most are small), but the variance between the magnitudes of a given channel across tokens is small (outlier channels are consistently large). Due to the persistence of outliers and the small variance inside each channel, if we could perform *per-channel* quantization [47] of the activation (i.e., using a different quantization step for each channel), the quantization error would be much smaller compared to *per-tensor* quantization, while *per-token* quantization helps little. In Table 3.1, we verify the assumption that *simulated* per-channel activation quantization successfully bridges the accuracy with the FP16 baseline, which echos the findings of Bondarenko, Nagel, and Blankevoort.

However, per-channel activation quantization does not map well to hardware-accelerated `GEMM` kernels, that rely on a sequence of operations executed at a high throughput (e.g., Tensor Core MMAs) and do not tolerate the insertion of instructions with a lower throughput (e.g., conversions or CUDA Core FMAs) in that sequence. In those kernels, scaling can only

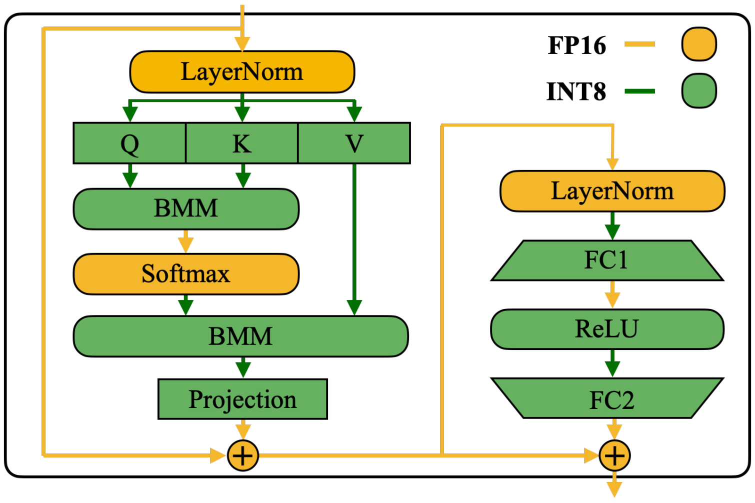


Figure 3.3. SmoothQuant's precision mapping for a Transformer block. All compute-intensive operators like linear layers and batched matmul (`BMMs`) use INT8 arithmetic.

be performed along the outer dimensions of the matrix multiplication (i.e., token dimension of activations T, output channel dimension of weights C_o, see Figure 3.1), which can be applied after the matrix multiplication finishes:

$$\mathbf{Y} = \text{diag}(\mathbf{\Delta}_{\mathbf{X}}^{\text{FP16}}) \cdot (\bar{\mathbf{X}}^{\text{INT8}} \cdot \bar{\mathbf{W}}^{\text{INT8}}) \cdot \text{diag}(\mathbf{\Delta}_{\mathbf{W}}^{\text{FP16}}) \tag{3.2}$$

Therefore, previous works all use per-token activation quantization for linear layers [16], [17], although they cannot address the difficulty of activation quantization (only slightly better than per-tensor).

3.3 SmoothQuant

Instead of per-channel activation quantization (which is infeasible), we propose to "smooth" the input activation by dividing it by a per-channel smoothing factor $\mathbf{s} \in \mathbb{R}^{C_i}$. To keep the mathematical equivalence of a linear layer, we scale the weights accordingly in the reversed direction:

$$\mathbf{Y} = (\mathbf{X}\text{diag}(\mathbf{s})^{-1}) \cdot (\text{diag}(\mathbf{s})\mathbf{W}) = \hat{\mathbf{X}}\hat{\mathbf{W}} \tag{3.3}$$

Considering input $\mathbf{X}$ is usually produced from previous linear operations (e.g., linear layers, layer norms, etc.), we can easily fuse the smoothing factor into previous layers' parameters *offline*, which doe not incur kernel call overhead from an extra scaling. For some other cases, when the input is from a residual add, we can add an extra scaling to the residual branch similar to [51].

Migrate the quantization difficulty from activations to weights. We aim to choose a per-channel smoothing factor $\mathbf{s}$ such that $\hat{\mathbf{X}} = \mathbf{X}\text{diag}(\mathbf{s})^{-1}$ is easy to quantize. To reduce the quantization error, we should *increase the effective quantization bits* for all the channels. The total effective quantization bits would be largest when all the channels have the same

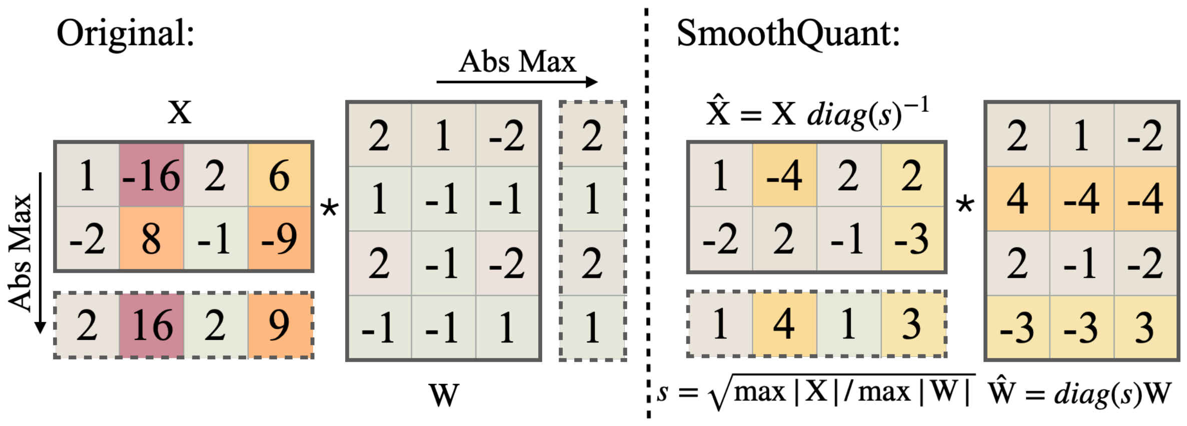


Figure 3.4. Main idea of SmoothQuant when α is 0.5. The smoothing factor s is obtained on calibration samples and the entire transformation is performed offline. At runtime, the activations are smooth without scaling.

maximum magnitude. Therefore, a straight-forward choice is $\mathbf{s}_j = \max(|\mathbf{X}_j|), j = 1, 2, ..., C_i$, where j corresponds to j-th input channel. This choice ensures that after the division, all the activation channels will have the same maximum value, which is easy to quantize. Note that the range of activations is dynamic; it varies for different input samples. Here, we estimate the scale of activations channels using calibration samples from the pre-training dataset [39]. However, this formula pushes *all* the quantization difficulties to the weights. We find that, in this case, the quantization errors would be large for the weights (outlier channels are migrated to weights now), leading to a large accuracy degradation (see Figure 3.8). On the other hand, we can also push all the quantization difficulty from weights to activations by choosing $\mathbf{s}_j = 1/\max(|\mathbf{W}_j|)$. Similarly, the model performance is bad due to the activation quantization errors. Therefore, we need to *split* the quantization difficulty between weights and activations so that they are both easy to quantize.

Here we introduce a hyper-parameter, migration strength α, to control how much difficulty we want to migrate from activation to weights, using the following equation:

$$\mathbf{s}_j = \max(|\mathbf{X}_j|)^\alpha / \max(|\mathbf{W}_j|)^{1-\alpha} \tag{3.4}$$

We find that for most of the models, e.g., all OPT [19] and BLOOM [20] models, $\alpha = 0.5$ is a well-balanced point to evenly split the quantization difficulty, especially when we are using the same quantizer for weights and activations (e.g., per-tensor, static quantization). The formula ensures that the weights and activations at the corresponding channel share a similar maximum value, thus sharing the same quantization difficulty. Figure 3.4 illustrates the smoothing transformation when we take $\alpha = 0.5$. For some other models where activation outliers are more significant (e.g., GLM-130B [21] has $\sim$30% outliers, which are more difficult for activation quantization), we can choose a larger α to migrate more quantization difficulty to weights (like 0.75).

Applying SmoothQuant to Transformer blocks. Linear layers take up most of the parameters and computation of LLM models. By default, we perform scale smoothing for the input activations of self-attention and feed-forward layers and quantize all linear layers

Table 3.2. Quantization setting of the baselines and SmoothQuant. All weight and activations use INT8 representations unless specified. For SmoothQuant, the efficiency **improves** from O1 to O3 (i.e., lower latency).

Method	Weight	Activation
W8A8	per-tensor	per-tensor dynamic
ZeroQuant	group-wise	per-token dynamic
`LLM.int8()`	per-channel	per-token dynamic+FP16
Outlier Suppression	per-tensor	per-tensor static
SmoothQuant-O1	per-tensor	per-token dynamic
SmoothQuant-O2	per-tensor	per-tensor dynamic
SmoothQuant-O3	per-tensor	per-tensor static

with W8A8. We also quantize `BMM` operators in the attention computation. We design a quantization flow for transformer blocks in Figure 3.3. We quantize the inputs and weights of compute-heavy operators like linear layers and `BMM` in attention layers with INT8, while keeping the activation as FP16 for other lightweight element-wise operations like ReLU, Softmax, and LayerNorm. Such a design helps us to balance accuracy and inference efficiency.

3.4 Experimental Results

3.4.1 Setups

Baselines. We compare with four baselines in the INT8 post-training quantization setting, i.e., without re-training of the model parameters: W8A8 naive quantization, ZeroQuant [17], `LLM.int8()` [16], and Outlier Suppression [51]. Since SmoothQuant is orthogonal to the quantization schemes, we provide gradually aggressive and efficient quantization levels from O1 to O3. The detailed quantization schemes of the baselines and SmoothQuant are shown in Table 3.2.

Models and datasets. We choose three families of LLMs to evaluate SmoothQuant: OPT [19], BLOOM [20], and GLM-130B [21]. We use seven zero-shot evaluation tasks: LAMBADA [61], HellaSwag [62], PIQA [63], WinoGrande [64], OpenBookQA [65], RTE [66], COPA [67], and one language modeling dataset WikiText [68] to evaluate the OPT and BLOOM models. We use MMLU [69], MNLI [70], QNLI [66] and LAMBADA to evaluate the GLM-130B model because some of the aforementioned benchmarks appear in the training set of GLM-130B. We use `lm-eval-harness`* to evaluate OPT and BLOOM models, and GLM-130B's official repo† for its own evaluation. Finally, we scale up our method to MT-NLG 530B [22] and for the first time enable the serving of a >500B model within a single node. Note that we focus on the *relative* performance change before and after quantization but not the absolute value.

*https://github.com/EleutherAI/lm-evaluation-harness
†https://github.com/THUDM/GLM-130B

Table 3.3. SmoothQuant maintains the accuracy of OPT-175B model after INT8 quantization, even with the most aggressive and most efficient O3 setting (Table 3.2). We extensively benchmark the performance on 7 zero-shot benchmarks (by reporting the average accuracy) and 1 language modeling benchmark (perplexity). *For ZeroQuant, we also tried leaving the input activation of self-attention in FP16 and quantizing the rest to INT8, which is their solution to the GPT-NeoX-20B. However, this does not solve the accuracy degradation of OPT-175B.

OPT-175B	LMBD	HS	PIQA	WNGD	OBQA	RTE	COPA	**Avg$\uparrow$**	**WikiText$\downarrow$**
FP16	74.7%	59.3%	79.7%	72.6%	34.0%	59.9%	88.0%	66.9%	10.99
W8A8	0.0%	25.6%	53.4%	50.3%	14.0%	49.5%	56.0%	35.5%	93080
ZeroQuant	0.0%*	26.0%	51.7%	49.3%	17.8%	50.9%	55.0%	35.8%	84648
`LLM.int8()`	74.7%	59.2%	79.7%	72.1%	34.2%	60.3%	87.0%	66.7%	11.10
Outlier Suppression	0.00%	25.8%	52.5%	48.6%	16.6%	53.4%	55.0%	36.0%	96151
SmoothQuant-O1	74.7%	59.2%	79.7%	71.2%	33.4%	58.1%	89.0%	66.5%	11.11
SmoothQuant-O2	75.0%	59.0%	79.2%	71.2%	33.0%	59.6%	88.0%	66.4%	11.14
SmoothQuant-O3	74.6%	58.9%	79.7%	71.2%	33.4%	59.9%	90.0%	66.8%	11.17

Activation smoothing. The migration strength $\alpha = 0.5$ is a general sweet spot for all the OPT and BLOOM models, and $\alpha = 0.75$ for GLM-130B since its activations are more difficult to quantize [21]. We get a suitable α by running a quick grid search on a subset of the Pile [71] validation set. To get the statistics of activations, we calibrate the smoothing factors and the static quantization step sizes *once* with 512 random sentences from the pre-training dataset Pile, and apply the same smoothed and quantized model for all downstream tasks. In this way, we can benchmark the generality and zero-shot performance of the quantized LLMs.

Implementation. We implement SmoothQuant with two backends: (1) PyTorch Huggingface[‡] for the proof of concept, and (2) FasterTransformer[§], as an example of a high-performance framework used in production environments. In both PyTorch Huggingface and FasterTransformer frameworks, we implement INT8 linear modules and the batched matrix multiplication (BMM) function with CUTLASS INT8 `GEMM` kernels. We simply replace the original floating point (FP16) linear modules and the `bmm` function with our INT8 kernels as the INT8 model.

3.4.2 Accurate Quantization

Results of OPT-175B. SmoothQuant can handle the quantization of very large LLMs, whose activations are more difficult to quantize. We study quantization on OPT-175B. As shown in Table 3.3, SmoothQuant can match the FP16 accuracy on all evaluation datasets with all quantization schemes. `LLM.int8()` can match the floating point accuracy because they use floating-point values to represent outliers, which leads to a large latency overhead (Table 3.11). The W8A8, ZeroQuant, and Outlier Suppression baselines produce nearly random results, indicating that naively quantizing the activation of LLMs will destroy the performance.

[‡]https://github.com/huggingface/transformers
[§]https://github.com/NVIDIA/FasterTransformer

Table 3.4. SmoothQuant works for different LLMs. We can quantize the 3 largest, openly available LLM models into INT8 without degrading the accuracy. For OPT-175B and BLOOM-176B, we show the average accuracy on WinoGrande, HellaSwag, PIQA, and LAMBADA. For GLM-130B we show the average accuracy on LAMBADA, MMLU, MNLI, and QNLI. *Accuracy is not column-wise comparable due to different datasets.

Method	OPT-175B	BLOOM-176B	GLM-130B*
FP16	71.6%	68.2%	73.8%
W8A8	32.3%	64.2%	26.9%
ZeroQuant	31.7%	67.4%	26.7%
`LLM.int8()`	71.4%	68.0%	73.8%
Outlier Suppression	31.7%	54.1%	63.5%
SmoothQuant-O1	**71.2%**	68.3%	**73.7%**
SmoothQuant-O2	71.1%	**68.4%**	72.5%
SmoothQuant-O3	71.1%	67.4%	72.8%

Results of different LLMs. SmoothQuant can be applied to various LLM designs. In Table 3.4, we show SmoothQuant can quantize all existing open LLMs beyond 100B parameters. Compared with the OPT-175B model, the BLOOM-176B model is easier to quantize: none of the baselines completely destroys the model; even the naive W8A8 per-tensor dynamic quantization only degrades the accuracy by 4%. The O1 and O2 levels of SmoothQuant successfully maintain the floating point accuracy, while the O3 level (per-tensor static) degrades the average accuracy by 0.8%, which we attribute to the discrepancy between the statically collected statistics and the real evaluation samples' activation statistics. On the contrary, the GLM-130B model is more difficult to quantize (which echos [21]). Nonetheless, SmoothQuant-O1 can match the FP16 accuracy, while SmoothQuant-O3 only degrades the accuracy by 1%, which significantly outperforms the baselines. Note that we clip the top 2% tokens when calibrating the static quantization step sizes for GLM-130B following [51]. Note that different model/training designs have different quantization difficulties, which we hope will inspire future research.

Results on LLMs of different sizes. SmoothQuant works not only for very large LLMs beyond 100B parameters, but it also works consistently for smaller LLMs. In Figure 3.5, we show that SmoothQuant can work on all scales of OPT models, matching the FP16 accuracy with INT8 quantization.

Results on Instruction-Tuned LLM Shown in Table 3.5, SmoothQuant also works on instruction-tuned LLMs. We test SmoothQuant on the OPT-IML-30B model using the WikiText-2 and LAMBADA datasets. Our results show that SmoothQuant successfully preserves model accuracy with W8A8 quantization, whereas the baselines fail to do so. SmoothQuant is a general method designed to balance the quantization difficulty for Transformer models. As the architecture of instruction-tuned LLMs is not fundamentally different from vanilla LLMs, and their pre-training processes are very similar, SmoothQuant is applicable to instruction-tuned LLMs as well.

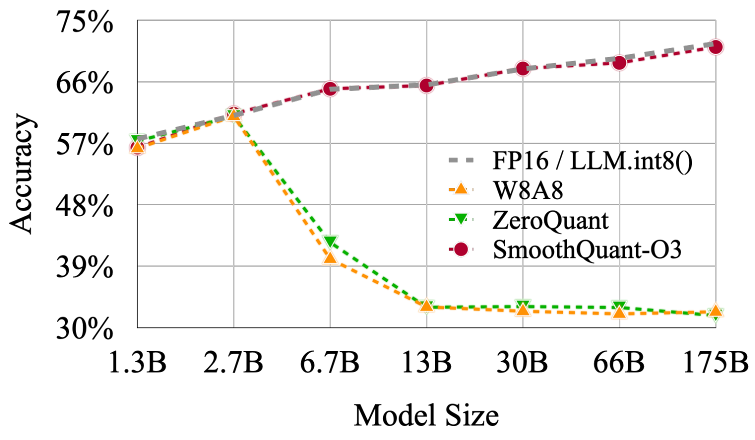


Figure 3.5. SmoothQuant-O3 (the most efficient setting, defined in Table 3.2) preserves the accuracy of OPT models across different scales when quantized to INT8. LLM.int8() requires mixed precision and suffers from slowing down.

Table 3.5. SmoothQuant's performance on the OPT-IML model.

OPT-IML-30B	LAMBADA ↑	WikiText ↓
FP16	69.12%	14.26
W8A8	4.21%	576.53
ZeroQuant	5.12%	455.12
LLM.int8()	69.14%	14.27
Outlier Suppression	0.00%	9485.62
SmoothQuant-O3	**69.77%**	**14.37**

Table 3.6. SmoothQuant can enable lossless W8A8 quantization for LLaMA models [5]. Results are perplexities on the WikiText-2 dataset with a sequence length of 512. We used per-token activation quantization and α=0.8 for SmoothQuant.

Wiki PPL↓	7B	13B	30B	65B
FP16	11.51	10.05	7.53	6.17
W8A8 SmoothQuant	11.56	10.08	7.56	6.20

Table 3.7. SmoothQuant can enable lossless W8A8 quantization for Llama-2 [6], Falcon [72], Mistral [73], and Mixtral [74] models. Results are perplexities on the WikiText-2 dataset with a sequence length of 2048. We used per-token activation quantization and per-channel weight quantization for SmoothQuant.

Model	Method	PPL	α
Llama-2-7B	FP16	5.474	
	W8A8 SQ	5.515	0.85
Llama-2-13B	FP16	4.950	
	W8A8 SQ	4.929	0.85
Llama-2-70B	FP16	3.320	
	W8A8 SQ	3.359	0.9
Falcon-7B	FP16	6.590	
	W8A8 SQ	6.629	0.6
Falcon-40B	FP16	5.228	
	W8A8 SQ	5.255	0.7
Mistral-7B	FP16	5.253	
	W8A8 SQ	5.277	0.8
Mixtral-8x7B	FP16	3.842	
	W8A8 SQ	3.893	0.8

Results on LLaMA models. LLaMA models are new open languange models with superior performance [5]. Through initial experiments, we find LLaMA models generally have less severe activation outlier issues compared to models like OPT and BLOOM. Nonetheless, SmoothQuant still works quite well for LLaMA models. We provide some initial results of LLaMA W8A8 quantization in Table 3.6. SmoothQuant enables W8A8 quantization at a negligible performance degradation.

Results on Llama-2, Falcon, Mistral, and Mixtral models. We apply SmoothQuant on several more recent LLMs using diverse architectures, such as Llama-2 [6], Falcon [72], Mistral [73], and Mixtral [74]—notably, the Mixtral model is a Mixture of Experts (MoE) model. The results, detailed in Table 3.7, demonstrate that SmoothQuant enables W8A8 quantization while maintaining performance with minimal loss across these varied architectures.

3.4.3 Speedup and Memory Saving

In this section, we show the measured speedup and memory saving of SmoothQuant-O3 integrated into PyTorch and FasterTransformer.

Context-stage: PyTorch Implementation. We measure the end-to-end latency of generating all hidden states for a batch of 4 sentences in one pass, i.e., the context stage latency. We record the (aggregated) peak GPU memory usage in this process. We only compare SmoothQuant with `LLM.int8()` because it is the only existing quantization method

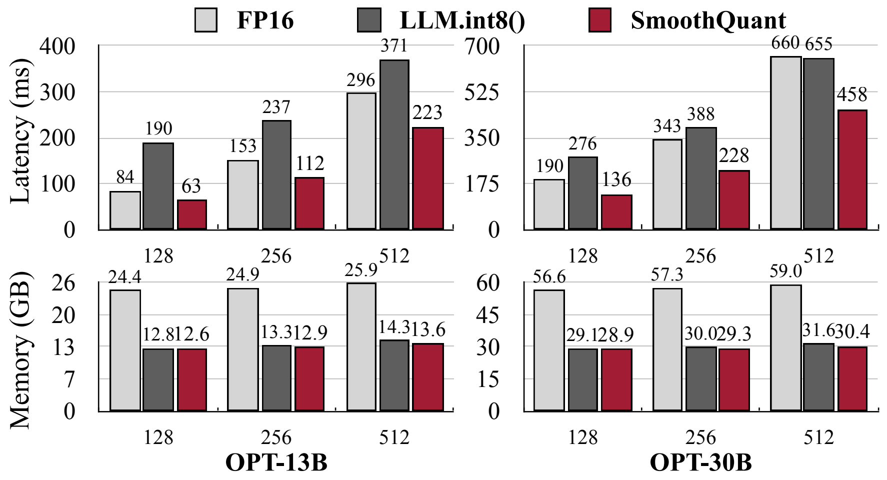


Figure 3.6. The PyTorch implementation of SmoothQuant-O3 achieves up to **1.51**× speedup and **1.96**× memory saving for OPT models on a single NVIDIA A100-80GB GPU, while `LLM.int8()` slows down the inference in most cases.

that can preserve LLM accuracy at all scales. Due to the lack of support for model parallelism in Huggingface, we only measure SmoothQuant's performance on a single GPU for the PyTorch implementation, so we choose OPT-6.7B, OPT-13B, and OPT-30B for evaluation. In the FasterTransformer library, SmoothQuant can seamlessly work with Tensor Parallelism [75] algorithm, so we test SmoothQuant on OPT-13B, OPT-30B, OPT-66B, and OPT-175B for both single and multi-GPU benchmarks. All our experiments are conducted on NVIDIA A100 80GB GPU servers.

In Figure 3.6, we show the inference latency and peak memory usage based on the PyTorch implementation. SmoothQuant is consistently faster than the FP16 baseline, getting a 1.51x speedup on OPT-30B when the sequence length is 256. We also see a trend that the larger the model, the more significant the acceleration. On the other hand, `LLM.int8()` is almost always slower than the FP16 baseline, which is due to the large overhead of the mixed-precision activation representation. In terms of memory, SmoothQuant and `LLM.int8()` can all nearly halve the memory usage of the FP16 model, while SmoothQuant saves slightly more memory because it uses fully INT8 `GEMM`s.

Context-stage: FasterTransformer Implementation. As shown in Figure 3.7 (top), compared to FasterTransformer's FP16 implementation of OPT, SmoothQuant-O3 can further reduce the execution latency of OPT-13B and OPT-30B by up to 1.56× when using a single GPU. This is challenging since FasterTransformer is already more than 3× faster compared to the PyTorch implementation for OPT-30B. Remarkably, for bigger models that have to be distributed across multiple GPUs, SmoothQuant achieves similar or even better latency using only *half* the number of GPUs (1 GPU instead of 2 for OPT-66B, 4 GPUs instead of 8 for OPT-175B). This could greatly lower the cost of serving LLMs. The amount of memory needed when using SmoothQuant-O3 in FasterTransformer is reduced by a factor of almost 2×, as shown on Figure 3.7 (bottom).

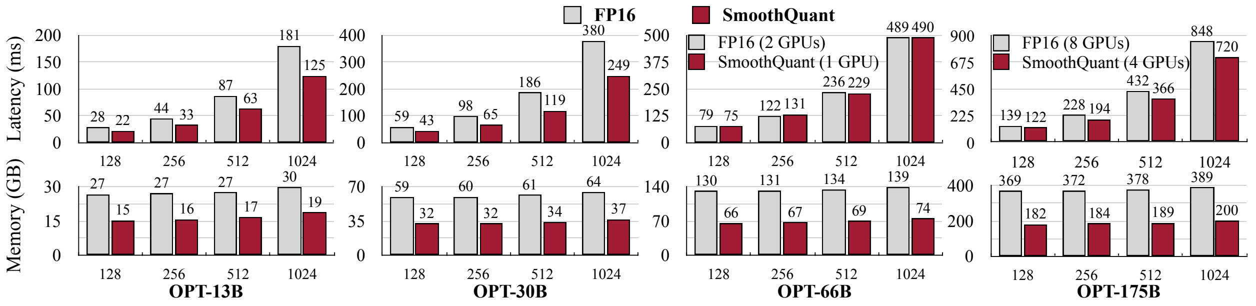


Figure 3.7. Inference latency (top) and memory usage (bottom) of the FasterTransformer implementation on NVIDIA A100-80GB GPUs. For smaller models, the latency can be significantly reduced with SmoothQuant-O3 by up to 1.56x compared to FP16. For the bigger models (OPT-66B and 175B), we can achieve similar or even faster inference using only **half** number of GPUs. Memory footprint is almost halved compared to FP16.

Table 3.8. SmoothQuant's performance in the decoding stage.

BS	SeqLen	Latency (ms)			Memory (GB)		
		FP16	Ours	Speedup (↑)	FP16	Ours	Saving (↑)
				OPT-30B (1 GPU)			
1	512	422	314	1.35×	57	30	1.91×
1	1024	559	440	1.27×	58	31	1.87×
16	512	2488	1753	1.42×	69	44	1.59×
16	1024	OOM	3947	-	OOM	61	-
				OPT-175B (8 GPUs)			
1	512	426	359	1.19×	44	23	1.87×
1	1024	571	475	1.20×	44	24	1.85×
16	512	2212	1628	1.36×	50	30	1.67×
16	1024	4133	3231	1.28×	56	37	1.52×

Decoding-stage. In Table 3.8, we show SmoothQuant can significantly accelerate the autoregressive decoding stage of LLMs. SmoothQuant constantly reduces the per-token decoding latency compared to FP16 (up to 1.42x speedup). Additionally, SmoothQuant halves the memory footprints for LLM inference, enabling the deployment of LLMs at a significantly lower cost.

Table 3.9. SmoothQuant can quantize MT-NLG 530B to W8A8 with negligible accuracy loss.

	LAMBADA	HellaSwag	PIQA	WinoGrande	Average
FP16	76.6%	62.1%	81.0%	72.9%	73.1%
INT8	77.2%	60.4%	80.7%	74.1%	73.1%

Table 3.10. When serving MT-NLG 530B, SmoothQuant can reduce the memory by half at a similar latency using *half* number of GPUs, which allows serving the 530B model within a single node.

SeqLen	Prec.	#GPUs	Latency	Memory
128	FP16	16	232ms	1040GB
	INT8	8	253ms	527GB
256	FP16	16	451ms	1054GB
	INT8	8	434ms	533GB
512	FP16	16	838ms	1068GB
	INT8	8	839ms	545GB
1024	FP16	16	1707ms	1095GB
	INT8	8	1689ms	570GB

Table 3.11. GPU Latency (ms) of different quantization schemes. The coarser the quantization scheme (from per-token to per-tensor, dynamic to static, O1 to O3, defined in Table 3.2), the lower the latency. SmoothQuant achieves lower latency compared to FP16 under all settings, while `LLM.int8()` is mostly slower. The batch size is 4.

Model	OPT-13B		OPT-30B	
Sequence Length	256	512	256	512
FP16	152.6	296.3	343.0	659.9
`LLM.int8()`	237.1	371.5	387.9	654.9
SmoothQuant-O1	124.5	243.3	246.7	490.7
SmoothQuant-O2	120.5	235.1	240.2	478.3
SmoothQuant-O3	112.1	223.1	227.6	458.4

3.4.4 Scaling Up: 530B Model Within a Single Node

We can further scale up SmoothQuant beyond 500B-level models, enabling efficient and accurate W8A8 quantization of MT-NLG 530B [22]. As shown in Table 3.9 and 3.10, SmoothQuant enables W8A8 quantization of the 530B model at a negligible accuracy loss. The reduced model size allows us to serve the model using half number of the GPUs (16 to 8) at a similar latency, enabling the serving of a >500B model within a single node (8×A100 80GB GPUs).

3.4.5 Ablation Study

Quantization schemes. Table 3.11 shows the inference latency of different quantization schemes based on our PyTorch implementation. We can see that the coarser the quantization granularity (from O1 to O3), the lower the latency. And static quantization can significantly accelerate inference compared with dynamic quantization because we no longer need to calculate the quantization step sizes at runtime. SmoothQuant is faster than FP16 baseline under all settings, while `LLM.int8()` is usually slower. We recommend using a coarser scheme

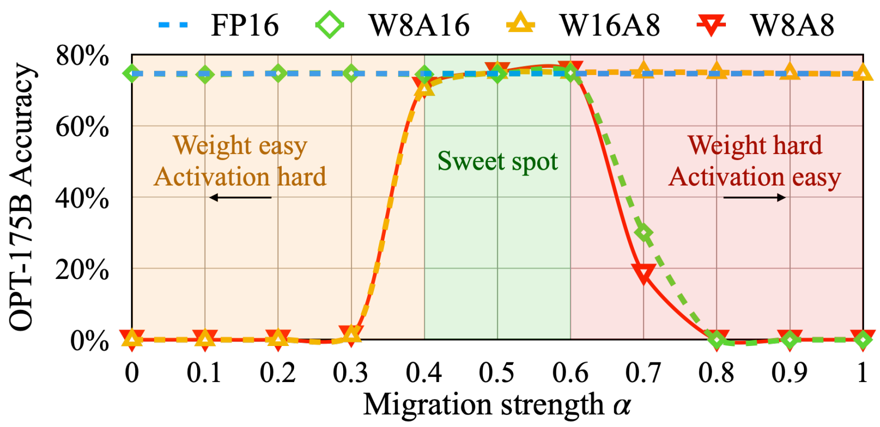


Figure 3.8. A suitable migration strength α (sweet spot) makes both activations and weights easy to quantize. If the α is too large, weights will be hard to quantize; if too small, activations will be hard to quantize.

if the accuracy permits.

Migration strength. We need to find a suitable migration strength α (see Equation 3.4) to balance the quantization difficulty of weights and activations. We ablate the effect of different α's on OPT-175B with LAMBADA in Figure 3.8. When α is too small ($<$0.4), the activations are hard to quantize; when α is too large ($>$0.6), the weights will be hard to quantize. Only when we choose α from the sweet spot region (0.4-0.6) can we get small quantization errors for both weights and activations, and maintain the model performance after quantization.

Chapter 4

StreamingLLM: Efficient Streaming Language Models with Attention Sinks

4.1 The Failure of Window Attention and Attention Sinks

While the window attention technique offers efficiency during inference, it results in an exceedingly high language modeling perplexity. Consequently, the model's performance is unsuitable for deployment in streaming applications. In this section, we use the concept of *attention sink* to explain the failure of window attention, serving as the inspiration behind StreamingLLM.

Identifying the Point of Perplexity Surge. Figure 4.1 shows the perplexity of language modeling on a 20K token text. It is evident that perplexity spikes when the text length surpasses the cache size, led by the exclusion of initial tokens. This suggests that the initial tokens, regardless of their distance from the predicted tokens, are crucial for maintaining the stability of LLMs.

Why do LLMs break when removing *initial* tokens' KV? We visualize attention maps from all layers and heads of the Llama-2-7B and models in Figure 1.4. We find that,

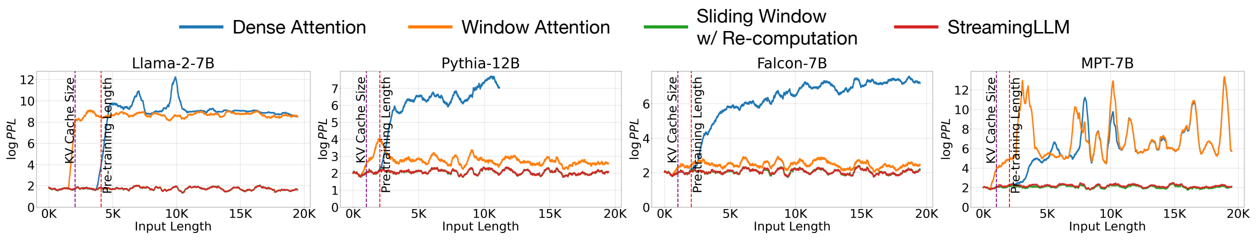


Figure 4.1. Language modeling perplexity on texts with 20K tokens across various LLM. Observations reveal consistent trends: (1) Dense attention fails once the input length surpasses the pre-training attention window size. (2) Window attention collapses once the input length exceeds the cache size, i.e., the initial tokens are evicted. (3) StreamingLLM demonstrates stable performance, with its perplexity nearly matching that of the sliding window with re-computation baseline.

beyond the bottom two layers, the model consistently focuses on the initial tokens across all layers and heads. The implication is clear: removing these initial tokens' KV will remove a considerable portion of the denominator in the SoftMax function (Equation 4.1) in attention computation. This alteration leads to a significant shift in the distribution of attention scores away from what would be expected in normal inference settings.

$$\text{SoftMax}(x)_i = \frac{e^{x_i}}{e^{x_1} + \sum_{j=2}^{N} e^{x_j}}, \quad x_1 \gg x_j, j \in 2, \ldots, N \tag{4.1}$$

There are two possible explanations for the importance of the initial tokens in language modeling: (1) Either their semantics are crucial, or (2) the model learns a bias towards their absolute position. To distinguish between these possibilities, we conduct experiments (Table 4.1), wherein the first four tokens are substituted with the linebreak token "\n". The observations indicate that the model still significantly emphasizes these initial linebreak tokens. Furthermore, reintroducing them restores the language modeling perplexity to levels comparable to having the original initial tokens. This suggests that the absolute position of the starting tokens, rather than their semantic value, holds greater significance.

LLMs attend to Initial Tokens as Attention Sinks. To explain why the model disproportionately focuses on initial tokens—regardless of their semantic relevance to language modeling, we introduce the concept of "*attention sink*". The nature of the SoftMax function (Equation 4.1) prevents all attended tokens from having zero values. This requires aggregating some information from other tokens across all heads in all layers, even if the current embedding has sufficient self-contained information for its prediction. Consequently, the model tends to dump unnecessary attention values to specific tokens. A similar observation has been made in the realm of quantization outliers [29], [76], leading to the proposal of SoftMax-Off-by-One [77] as a potential remedy.

Table 4.1. Window attention has poor performance on long text. The perplexity is restored when we reintroduce the initial four tokens alongside the recent 1020 tokens (4+1020). Substituting the original four initial tokens with linebreak tokens "\n" (4"\n"+1020) achieves comparable perplexity restoration. Cache config x+y denotes adding x initial tokens with y recent tokens. Perplexities are measured on the first book (65K tokens) in the PG19 test set.

Llama-2-13B	PPL (↓)
0 + 1024 (Window)	5158.07
4 + 1020	5.40
4"\n"+1020	5.60

Table 4.2. Effects of reintroduced initial token numbers on StreamingLLM. (1) Window attention (0+y) has a drastic increase in perplexity. (2) Introducing one or two initial tokens doesn't fully restore model perplexity, showing that the model doesn't solely use the first token as the attention sink. (3) Introducing four initial tokens generally suffices; further additions have diminishing returns. Cache config x+y denotes adding x initial tokens to y recent tokens. Perplexities are evaluated on 400K tokens in the concatenated PG19 test set.

Cache Config	0+2048	1+2047	2+2046	4+2044	8+2040
Falcon-7B	17.90	12.12	12.12	12.12	12.12
MPT-7B	460.29	14.99	15.00	14.99	14.98
Pythia-12B	21.62	11.95	12.09	12.09	12.02
Cache Config	**0+4096**	**1+4095**	**2+4094**	**4+4092**	**8+4088**
Llama-2-7B	3359.95	11.88	10.51	9.59	9.54

Why do various autoregressive LLMs, such as Llama-2, MPT, Falcon, and Pythia, consistently focus on *initial tokens* as their attention sinks, rather than other tokens? Our explanation is straightforward: Due to the sequential nature of autoregressive language modeling, initial tokens are visible to all subsequent tokens, while later tokens are only visible to a limited set of subsequent tokens. As a result, initial tokens are more easily trained to serve as attention sinks, capturing unnecessary attention.

We've noted that LLMs are typically trained to utilize multiple initial tokens as attention sinks rather than just one. As illustrated in Figure 4.2, the introduction of four initial tokens, as attention sinks, suffices to restore the LLM's performance. In contrast, adding just one or two doesn't achieve full recovery. We believe this pattern emerges because these models didn't include a consistent starting token across all input samples during pre-training. Although Llama-2 does prefix each paragraph with a "<s>" token, it's applied before text chunking, resulting in a mostly random token occupying the zeroth position. This lack of a uniform starting token leads the model to use several initial tokens as attention sinks. We hypothesize that by incorporating a stable learnable token at the start of all training samples, it could singularly act as a committed attention sink, eliminating the need for multiple initial tokens to ensure consistent streaming. We will validate this hypothesis in Section 4.3.

4.2 Rolling KV Cache with Attention Sinks

To enable LLM streaming in already trained LLMs, we propose a straightforward method that can recover window attention's perplexity without any model finetuning. Alongside the current sliding window tokens, we reintroduce a few starting tokens' KV in the attention computation. The KV cache in StreamingLLM can be conceptually divided into two parts, as illustrated in Figure 4.2: (1) Attention sinks (four initial tokens) stabilize the attention

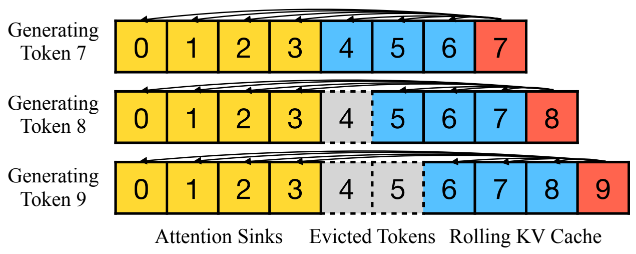


Figure 4.2. The KV cache of StreamingLLM.

Table 4.3. Comparison of vanilla attention with prepending a zero token and a learnable sink token during pre-training. To ensure stable streaming perplexity, the vanilla model requires several initial tokens. While Zero Sink shows a slight improvement, it still needs other initial tokens. Conversely, the model trained with a learnable Sink Token shows stable streaming perplexity with only the sink token added. Cache config x+y denotes adding x initial tokens with y recent tokens. Perplexity is evaluated on the first sample in the PG19 test set.

Cache Config	0+1024	1+1023	2+1022	4+1020
Vanilla	27.87	18.49	18.05	18.05
Zero Sink	29214	19.90	18.27	18.01
Learnable Sink	1235	**18.01**	18.01	18.02

computation; 2) Rolling KV Cache retains the most recent tokens, crucial for language modeling. StreamingLLM' design is versatile and can be seamlessly incorporated into any autoregressive language model that employs relative positional encoding, such as RoPE [53] and ALiBi [33].

When determining the relative distance and adding positional information to tokens, StreamingLLM focuses on positions *within the cache* rather than those *in the original text*. This distinction is crucial for StreamingLLM's performance. For instance, if the current cache (Figure 4.2) has tokens [0, 1, 2, 3, 6, 7, 8] and is in the process of decoding the 9th token, the positions assigned are [0, 1, 2, 3, 4, 5, 6, 7], rather than the positions in the original text, which would be [0, 1, 2, 3, 6, 7, 8, 9].

For encoding like RoPE, we cache the Keys of tokens *prior to* introducing the rotary transformation. Then, we apply position transformation to the keys in the rolling cache at each decoding phase. On the other hand, integrating with ALiBi is more direct. Here, the contiguous linear bias is applied instead of a 'jumping' bias to the attention scores. This method of assigning positional embedding within the cache is crucial to StreamingLLM's functionality, ensuring that the model operates efficiently even beyond its pre-training attention window size.

4.3 Pre-Training LLMs with Attention Sinks

As elaborated in Section 4.1, a significant reason for the model's excessive attention to multiple initial tokens is the absence of a designated sink token to offload excessive attention scores. Due to this, the model inadvertently uses globally visible tokens, primarily the initial ones, as attention sinks. A potential remedy can be the intentional inclusion of a global trainable attention sink token, denoted as a “Sink Token”, which would serve as a repository for unnecessary attention scores. Alternatively, replacing the conventional SoftMax function with a variant like SoftMax-off-by-One [77],

$$\text{SoftMax}_1(x)_i = \frac{e^{x_i}}{1 + \sum_{j=1}^{N} e^{x_j}}, \tag{4.2}$$

which does not require the attention scores on all contextual tokens to sum up to one, may also be effective. Note that SoftMax_1 is equivalent to prepending a token with an all-zero Key and Value features in the attention computation. We denote this method as “Zero Sink” to fit our framework.

For validation, we pre-train three language models with 160 million parameters from scratch under identical settings. The first model utilizes the standard SoftMax attention (Vanilla), the second replaced the regular attention mechanism with SoftMax_1 (Zero Sink), and one prepending a learnable placeholder token (Sink Token) in all training samples. As shown in Table 4.3, while the zero sink alleviates the attention sink problem to some extent, the model still relies on other initial tokens as attention sinks. Introducing a sink token is highly effective in stabilizing the attention mechanism. Simply pairing this sink token with recent tokens sufficiently anchors the model’s performance, and the resulting evaluation perplexity is even marginally improved. Given these findings, we recommend training future LLMs with a sink token in all samples to optimize streaming deployment.

4.4 Experimental Results

We evaluate StreamingLLM using four prominent recent model families: Llama-2 [6], MPT [78], PyThia [79], and Falcon [80]. Notably, Llama-2, Falcon, and Pythia incorporate RoPE [53], whereas MPT employs ALiBi [33] — two of the most influential position encoding techniques in recent research. Our diverse model selection ensures the validity and robustness of our findings. We benchmark StreamingLLM against established baselines such as dense attention, window attention, and the sliding window approach with re-computation. In all subsequent experiments with StreamingLLM, we default to using four initial tokens as attention sinks unless stated otherwise.

4.4.1 Language Modeling on Long Texts Across LLM Families and Scales

We firstly evaluate StreamingLLM’s language modeling perplexity using the concatenated PG19 [81] test set, which contains 100 long books. For Llama-2 models, the cache size is set at 2048, while for Falcon, Pythia, and MPT models, it’s set at 1024. This is half the pre-training window size chosen to enhance visualization clarity.

Figure 4.1 illustrates that StreamingLLM can match the oracle baseline (sliding window with re-computation) in terms of perplexity on texts spanning 20K tokens. Meanwhile, the dense attention technique fails when the input length exceeds its pre-training window, and

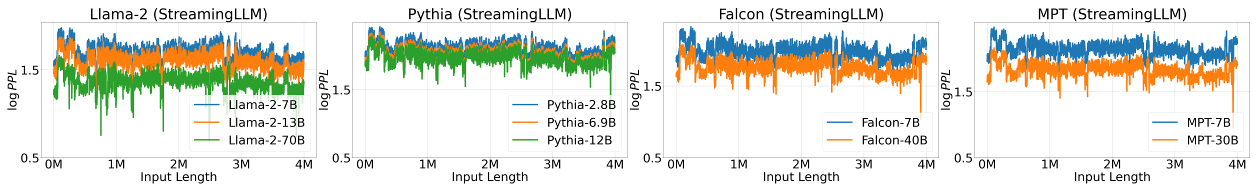


Figure 4.3. Language modeling perplexity of StreamingLLM on super long texts with 4 million tokens across various LLM families and scales. The perplexity remains stable throughout. We use the concatenated test set of PG19 (100 books) to perform language modeling, with perplexity fluctuations due to book transitions.

the window attention technique struggles when the input length surpasses the cache size, leading to the eviction of the initial tokens. In Figure 4.3, we further substantiate that StreamingLLM can reliably handle exceptionally extended texts, encompassing more than 4 million tokens, across a spectrum of model families and scales. This includes Llama-2-[7,13,70]B, Falcon-[7,40]B, Pythia-[2.8,6.9,12]B, and MPT-[7,30]B.

4.4.2 Results of Pre-Training with a Sink Token

To validate our suggestion that introducing a sink token to all pre-training samples improves streaming LLMs, we trained two language models, each with 160 million parameters, under identical conditions. While one model adhered to the original training settings, the other incorporated a sink token at the start of every training sample. Our experiments employed the Pythia-160M [79] codebase and followed its training recipe. We train the models on an 8xA6000 NVIDIA GPU server using the deduplicated Pile [82] dataset. Apart from reducing the training batch size to 256, we retained all Pythia training configurations, including learning rate schedules, model initialization, and dataset permutations. Both models were trained for 143,000 steps.

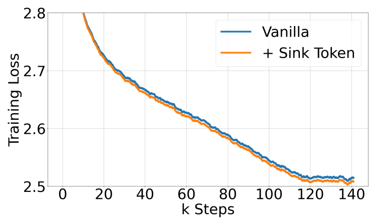


Figure 4.4. Pre-training loss curves of models w/ and w/o sink tokens. Two models have a similar convergence trend.

Table 4.4. Zero-shot accuracy (in %) across 7 NLP benchmarks, including ARC-[Challenge, Easy], HellaSwag, LAMBADA, OpenbookQA, PIQA, and Winogrande. The inclusion of a sink token during pre-training doesn't harm the model performance.

Methods	ARC-c	ARC-e	HS	LBD	OBQA	PIQA	WG
Vanilla	18.6	45.2	29.4	39.6	16.0	62.2	50.1
+Sink Token	**19.6**	**45.6**	**29.8**	**39.9**	**16.6**	**62.6**	**50.8**

Convergence and Normal Model Performance. Including a sink token during pre-training has no negative impact on model convergence and subsequent performance on a range of NLP benchmarks. As depicted in Figure 4.4, models trained with a sink token exhibit similar convergence dynamics compared to their vanilla counterparts. We evaluate

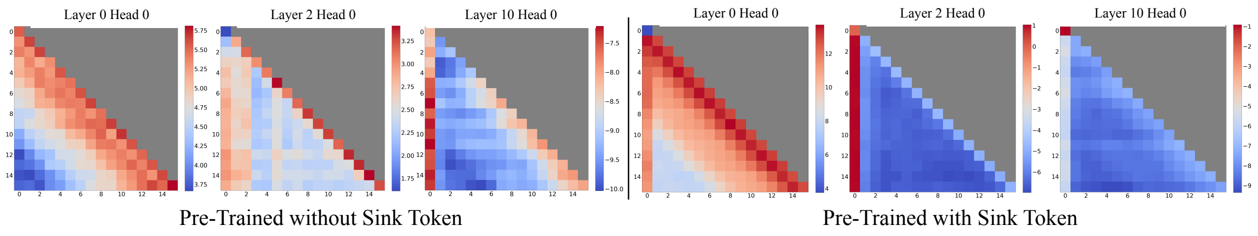


Figure 4.5. Visualization of average attention logits over 256 sentences, each 16 tokens long, comparing models pre-trained without (left) and with (right) a sink token. Both maps show the same layers and heads. Key observations: (1) Without a sink token, models show local attention in lower layers and increased attention to initial tokens in deeper layers. (2) With a sink token, there is clear attention directed at it across all layers, effectively collecting redundant attention. (3) With the presence of the sink token, less attention is given to other initial tokens, supporting the benefit of designating the sink token to enhance the streaming performance.

the two models on seven diverse NLP benchmarks, including ARC-[Challenge, Easy] [83], HellaSwag [62], LAMBADA [61], OpenbookQA [65], PIQA [84], and Winogrande [64]. As shown in Table 4.4, the model pre-trained with a sink token performs similarly to that trained using the vanilla approach.

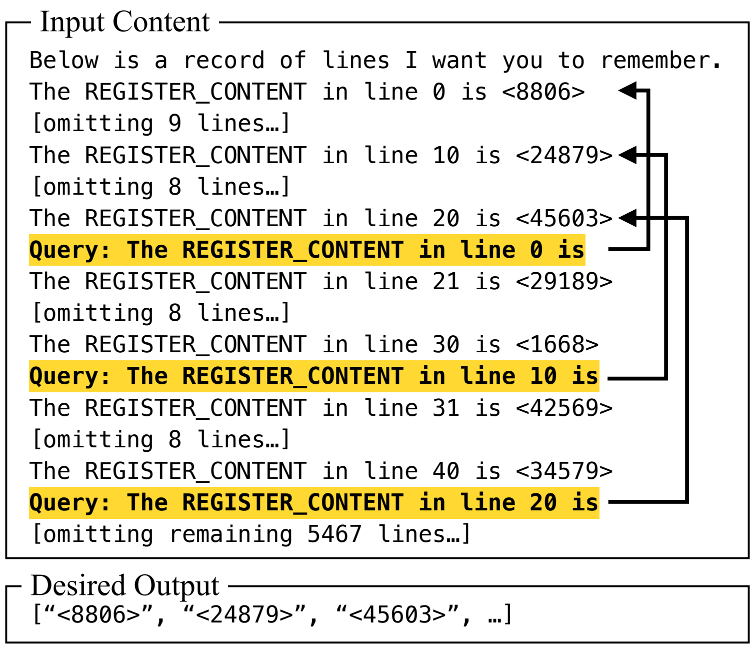
Streaming Performance. As illustrated in Table 4.3, the streaming perplexities differ between models trained using traditional methods and those augmented with a sink token. Remarkably, the vanilla model requires the addition of multiple tokens as attention sinks to maintain stable streaming perplexity. In contrast, the model trained with a sink token achieves satisfactory streaming performance using just the sink token.

Attention Visualization. Figure 4.5 contrasts attention maps for models pre-trained with and without a sink token. The model without the sink token, similar to Llama-2-7B (Figure 1.4), shows early-layer local attention and deeper-layer focus on initial tokens. In contrast, models trained with a sink token consistently concentrate on the sink across layers and heads, indicating an effective attention offloading mechanism. This strong focus on the sink, with reduced attention to other initial tokens, explains the sink token's efficacy in enhancing model's streaming performance.

4.4.3 Results on Streaming Question Answering with Instruction-tuned Models

To show StreamingLLM's real-world applicability, we emulate multi-round question-answering using instruction-tuned LLMs, commonly used in real-world scenarios.

We first concatenate all question-answer pairs from the ARC-[Challenge, Easy] datasets, feed the continuous stream to Llama-2-[7,13,70]B-Chat models, and assess model completions at each answer position using an exact match criterion. As table 4.5 indicates, dense attention results in Out-of-Memory (OOM) errors, showing it unsuitable for this setting. While the window attention method works efficiently, it exhibits low accuracy due to random outputs when the input length exceeds the cache size. Conversely, StreamingLLM excels



```
Below is a record of lines I want you to remember.
The REGISTER_CONTENT in line 0 is <8806>
[omitting 9 lines…]
The REGISTER_CONTENT in line 10 is <24879>
[omitting 8 lines…]
The REGISTER_CONTENT in line 20 is <45603>
Query: The REGISTER_CONTENT in line 0 is
The REGISTER_CONTENT in line 21 is <29189>
[omitting 8 lines…]
The REGISTER_CONTENT in line 30 is <1668>
Query: The REGISTER_CONTENT in line 10 is
The REGISTER_CONTENT in line 31 is <42569>
[omitting 8 lines…]
The REGISTER_CONTENT in line 40 is <34579>
Query: The REGISTER_CONTENT in line 20 is
[omitting remaining 5467 lines…]
```

```
["<8806>", "<24879>", "<45603>", …]
```

Figure 4.6. The first sample in StreamEval.

by efficiently handling the streaming format, aligning with the one-shot, sample-by-sample baseline accuracy.

Highlighting a more fitting scenario for StreamingLLM, we introduce a dataset, StreamEval, inspired by the LongEval [59] benchmark. As depicted in Figure 4.6, diverging from LongEval's single query over a long-span setup, we query the model every 10 lines of new information. Each query's answer is consistently 20 lines prior, reflecting real-world instances where questions typically pertain to recent information. As illustrated in Figure 4.7, LLMs employing StreamingLLM maintain reasonable accuracy even as input lengths approach 120K tokens. In contrast, both dense and window attention fail at the pre-training text length and the KV cache size, respectively. Additionally, we utilize two context-extended models, LongChat-7b-v1.5-32k [59] and Llama-2-7B-32K-Instruct [85], to show that StreamingLLM can complement

Table 4.5. Accuracy (in %) on the ARC-[Easy, Challenge] datasets. Questions were concatenated and answered in a streaming manner to mimic a real-world chat setting. The dense baseline fails due to Out-of-Memory (OOM) errors. Window attention has poor accuracy. StreamingLLM has comparable results with the one-shot sample-by-sample baseline. Window attention and StreamingLLM use cache sizes of 1024.

Model	Llama-2-7B-Chat		Llama-2-13B-Chat		Llama-2-70B-Chat	
Dataset	Arc-E	Arc-C	Arc-E	Arc-C	Arc-E	Arc-C
One-shot	71.25	53.16	78.16	63.31	91.29	78.50
Dense	OOM					
Window	3.58	1.39	0.25	0.34	0.12	0.32
StreamingLLM	71.34	55.03	80.89	65.61	91.37	80.20

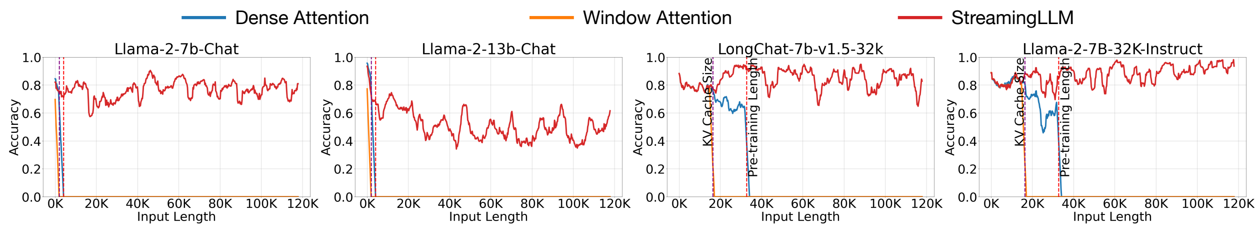


Figure 4.7. Performance on the StreamEval benchmark. Accuracies are averaged over 100 samples.

Table 4.6. Effects of cache size on StreamingLLM's performance. Increasing the cache size in StreamingLLM doesn't consistently yield a decrease in perplexity, showing these models may not fully utilize the provided context. Cache config x+y denotes adding x initial tokens with y recent tokens. Perplexity is evaluated on 400K tokens in the concatenated PG19 test set.

Cache	4+252	4+508	4+1020	4+2044
Falcon-7B	13.61	12.84	**12.34**	12.84
MPT-7B	**14.12**	14.25	14.33	14.99
Pythia-12B	13.17	12.52	**12.08**	12.09

Cache	4+508	4+1020	4+2044	4+4092
Llama-2-7B	9.73	9.32	**9.08**	9.59

context extension techniques. Within StreamingLLM, context extension means broadening the maximum cache size of streaming LLMs, enabling the capture of broader local information.

4.4.4 Ablation Studies

Numbers of Initial Tokens. In Table 4.2, we ablate the effect of adding varying numbers of initial tokens with recent tokens on the streaming perplexity. The results show the insufficiency of introducing merely one or two initial tokens, whereas a threshold of four initial tokens appears enough, with subsequent additions contributing marginal effects. This result justifies our choice of introducing 4 initial tokens as attention sinks in StreamingLLM.

Cache Sizes. In Table 4.6, we evaluate cache size's impact on StreamingLLM's perplexity. Contrary to intuition, increasing the cache size doesn't consistently lower the language modeling perplexity. This inconsistency shows a potential limitation where these models might not maximize the utility of the entire context they receive. Future research efforts should target enhancing these models' capabilities to utilize extensive contexts better.

4.4.5 Efficency Results

We benchmark StreamingLLM's decoding latency and memory usage against the sliding window with re-computation, which is the only baseline with acceptable quality. Both methods are implemented using the Huggingface Transformers library [86] and tested on

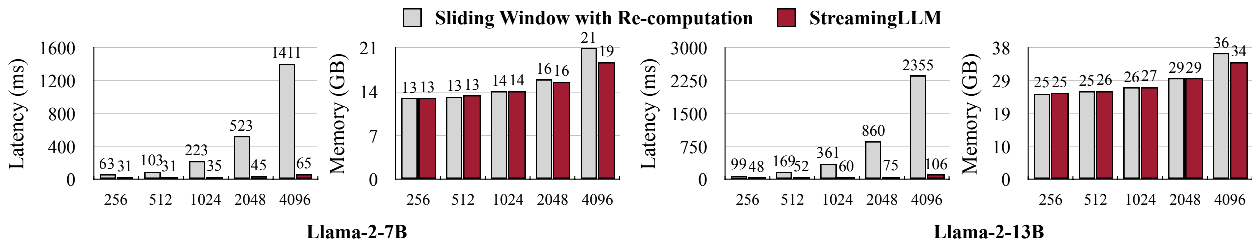


Figure 4.8. Comparison of per-token decoding latency and memory usage between the sliding window approach with re-computation baseline and StreamingLLM, plotted against the cache size (attention window size) on the X-axis. StreamingLLM delivers a remarkable speedup of up to 22.2× per token and retains a memory footprint similar to the re-computation baseline.

a single NVIDIA A6000 GPU using the Llama-2-7B and Llama-2-13B models. As shown in Figure 4.8, as the cache size increases, StreamingLLM's decoding speed has a linear growth. The sliding window with re-computation baseline has a quadratic rise in decoding latency. Thus, StreamingLLM achieves an impressive speedup, reaching up to 22.2× per token. Despite its reduced latency, StreamingLLM sustains a memory footprint consistent with the re-computation baseline.

Chapter 5

Conclusion

Large language models (LLMs) have demonstrated exceptional performance across various natural language processing tasks. However, their deployment is both compute- and memory-intensive, posing significant challenges for widespread and efficient use. This thesis has explored two primary solutions to address these challenges: SmoothQuant and StreamingLLM.

5.1 SmoothQuant: Efficient Quantization for LLMs

Quantization is a proven method to reduce memory usage and accelerate inference in neural networks. However, existing quantization methods for LLMs often struggle to maintain accuracy and hardware efficiency simultaneously. We introduced SmoothQuant, a training-free, accuracy-preserving, and general-purpose post-training quantization (PTQ) solution designed to enable 8-bit weight, 8-bit activation (W8A8) quantization for LLMs.

SmoothQuant is based on the observation that while weights are relatively easy to quantize, activations are not, primarily due to the presence of outliers. To address this, SmoothQuant employs a mathematically equivalent transformation that migrates the quantization difficulty from activations to weights, smoothing the activation outliers in the process. This enables effective INT8 quantization for both weights and activations across all matrix multiplications in LLMs, including models such as OPT, BLOOM, GLM, MT-NLG, Llama-1/2, Falcon, Mistral, and Mixtral.

Our experimental results demonstrate that SmoothQuant achieves up to $1.56\times$ speedup and $2\times$ memory reduction for LLMs, with negligible loss in accuracy. Moreover, SmoothQuant allows the serving of a 530B parameter LLM within a single node, significantly reducing hardware costs and making LLMs more accessible.

5.2 StreamingLLM: Enabling LLMs for Infinite-Length Inputs

Deploying LLMs in streaming applications, such as multi-round dialogue systems where long interactions are expected, presents additional challenges. These include extensive memory consumption due to caching previous tokens' Key and Value states (KV) during decoding, and the inability of popular LLMs to generalize beyond the training sequence length.

Window attention, which caches only the most recent KVs, seems a natural solution but fails when text length exceeds the cache size. Through our research, we identified an interesting phenomenon called *attention sink*, where initial tokens with high attention scores act as sinks, recovering the performance of window attention despite their lack of semantic importance.

Building on this insight, we developed StreamingLLM, an efficient framework that enables LLMs trained with a finite-length attention window to handle infinite sequence lengths without any fine-tuning. Our approach retains the KV states of a few initial tokens to maintain stable performance. We demonstrated that StreamingLLM can enable models such as Llama-2, MPT, Falcon, and Pythia to perform stable and efficient language modeling with up to 4 million tokens and beyond. Additionally, introducing a placeholder token as a dedicated attention sink during pre-training further enhances streaming deployment efficiency.

In streaming settings, StreamingLLM outperforms the sliding window recomputation baseline by up to 22.2× speedup, showcasing its potential for real-world applications.

5.3 Research Impact

The advancements presented in SmoothQuant and StreamingLLM have significantly enhanced the efficiency and applicability of large language models (LLMs), garnering notable impact in both the research community and industry. SmoothQuant was accepted to the International Conference on Machine Learning (ICML) 2023, while StreamingLLM was published at the International Conference on Learning Representations (ICLR) 2024. Both works have received positive feedback from researchers and practitioners, who have shown substantial interest in adopting these techniques and pursuing further research.

As of the thesis submission date, SmoothQuant has been cited 295 times, and StreamingLLM has been cited 97 times. SmoothQuant's straightforward yet effective approach to handling activation outliers has spurred further research on LLM quantization, particularly in addressing outliers [87]–[89]. StreamingLLM's discovery of attention sinks has been recognized as a universal phenomenon across various transformers [90], not only in language models but also in vision-language models [91]. This concept has not only contributed to the acceleration and compression of transformers [92]–[94], but has also deepened the theoretical understanding of transformer operations [95], [96].

In industry, SmoothQuant has been integrated into NVIDIA's FasterTransformer and TensorRT-LLM, Intel's Neural Compressor and Q8-Chat LLM. StreamingLLM has been adopted by NVIDIA's TensorRT-LLM, Intel's Extension for Transformers, Huggingface's Transformers library, and MLC-LLM for enabling continuous chat on iPhone. Both projects have been made available as open-source, with SmoothQuant receiving 1K GitHub stars and StreamingLLM receiving 6.3K stars up to the submission date. StreamingLLM also garnered significant media coverage, including articles in VentureBeat, MIT News, and The Daily Beast.

By addressing the computational and memory challenges associated with LLMs, this thesis paves the way for more accessible and efficient deployment of large language models across a variety of applications. This democratizes their use and enables broader innovation in the field of artificial intelligence.

References

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

[2] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[3] S. Zhang, S. Roller, N. Goyal, *et al.*, *Opt: Open pre-trained transformer language models*, 2022. arXiv: 2205.01068 [cs.CL].

[4] OpenAI, *Gpt-4 technical report*, 2023. arXiv: 2303.08774 [cs.CL].

[5] 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.

[6] H. Touvron, L. Martin, K. Stone, *et al.*, *Llama 2: Open foundation and fine-tuned chat models*, 2023. arXiv: 2307.09288 [cs.CL].

[7] J. Schulman, B. Zoph, C. Kim, J. Hilton, J. Menick, J. Weng, J. F. C. Uribe, L. Fedus, L. Metz, M. Pokorny, *et al.*, “Chatgpt: Optimizing language models for dialogue,” *OpenAI blog*, 2022.

[8] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, *Stanford alpaca: An instruction-following llama model*, https://github.com/tatsu-lab/stanford_alpaca, 2023.

[9] W.-L. Chiang, Z. Li, Z. Lin, *et al.*, *Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality*, Mar. 2023. URL: https://lmsys.org/blog/2023-03-30-vicuna/.

[10] T. Goyal and G. Durrett, “Evaluating factuality in generation with dependency-level entailment,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Online: Association for Computational Linguistics, 2020.

[11] T. Zhang, F. Ladhak, E. Durmus, P. Liang, K. McKeown, and T. B. Hashimoto, *Benchmarking large language models for news summarization*, 2023. arXiv: 2301.13848 [cs.CL].

[12] M. Chen, J. Tworek, H. Jun, *et al.*, *Evaluating large language models trained on code*, 2021. arXiv: 2107.03374 [cs.LG].

[13] B. Rozière, J. Gehring, F. Gloeckle, *et al.*, *Code Llama: Open foundation models for code*, 2023. arXiv: 2308.12950 [cs.CL].

[14] E. Kamalloo, N. Dziri, C. L. A. Clarke, and D. Rafiei, *Evaluating open-domain question answering in the era of large language models*, 2023. arXiv: 2305.06984 [cs.CL].

[15] T. Brown, B. Mann, N. Ryder, *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901. URL: https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[16] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "Llm.int8(): 8-bit matrix multiplication for transformers at scale," *arXiv preprint arXiv:2208.07339*, 2022.

[17] Z. Yao, R. Y. Aminabadi, M. Zhang, X. Wu, C. Li, and Y. He, *Zeroquant: Efficient and affordable post-training quantization for large-scale transformers*, 2022. DOI: 10.48550/ARXIV.2206.01861. URL: https://arxiv.org/abs/2206.01861.

[18] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT 2019*, Association for Computational Linguistics, 2019, pp. 4171–4186.

[19] S. Zhang, S. Roller, N. Goyal, *et al.*, *Opt: Open pre-trained transformer language models*, 2022. DOI: 10.48550/ARXIV.2205.01068. URL: https://arxiv.org/abs/2205.01068.

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

[21] A. Zeng, X. Liu, Z. Du, Z. Wang, H. Lai, M. Ding, Z. Yang, Y. Xu, W. Zheng, X. Xia, *et al.*, "Glm-130b: An open bilingual pre-trained model," *arXiv preprint arXiv:2210.02414*, 2022.

[22] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, *et al.*, "Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model," *arXiv preprint arXiv:2201.11990*, 2022.

[23] S. Chen, S. Wong, L. Chen, and Y. Tian, *Extending context window of large language models via positional interpolation*, arXiv: 2306.15595, 2023. arXiv: 2306.15595 [cs.CL].

[24] kaiokendev, *Things I'm learning while training superhot.* 2023. URL: https://kaiokendev.github.io/til#extending-context-to-8k.

[25] B. Peng, J. Quesnelle, H. Fan, and E. Shippole, *Yarn: Efficient context window extension of large language models*, 2023. arXiv: 2309.00071 [cs.CL].

[26] T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré, *FlashAttention: Fast and memory-efficient exact attention with IO-awareness*, arXiv:2205.14135, 2022.

[27] T. Dao, "FlashAttention-2: Faster attention with better parallelism and work partitioning," 2023.

[28] R. Pope, S. Douglas, A. Chowdhery, J. Devlin, J. Bradbury, A. Levskaya, J. Heek, K. Xiao, S. Agrawal, and J. Dean, "Efficiently scaling transformer inference," *arXiv preprint arXiv:2211.05102*, 2022.

[29] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "SmoothQuant: Accurate and efficient post-training quantization for large language models," in *Proceedings of the 40th International Conference on Machine Learning*, 2023.

[30] S. Anagnostidis, D. Pavllo, L. Biggio, L. Noci, A. Lucchi, and T. Hofmann, *Dynamic context pruning for efficient and interpretable autoregressive transformers*, 2023. arXiv: 2305.15805 [cs.CL].

[31] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," *HPCA*, 2021.

[32] Z. Zhang, Y. Sheng, T. Zhou, *et al.*, *H_2o: Heavy-hitter oracle for efficient generative inference of large language models*, 2023. arXiv: 2306.14048 [cs.LG].

[33] O. Press, N. Smith, and M. Lewis, "Train short, test long: Attention with linear biases enables input length extrapolation," in *International Conference on Learning Representations*, 2022. URL: https://openreview.net/forum?id=R8sQPpGCv0.

[34] I. Beltagy, M. E. Peters, and A. Cohan, *Longformer: The long-document transformer*, arXiv:2004.05150, 2020.

[35] J. W. Rae, S. Borgeaud, T. Cai, K. Millican, J. Hoffmann, F. Song, J. Aslanides, S. Henderson, R. Ring, S. Young, *et al.*, "Scaling language models: Methods, analysis & insights from training gopher," *arXiv preprint arXiv:2112.11446*, 2021.

[36] N. Du, Y. Huang, A. M. Dai, S. Tong, D. Lepikhin, Y. Xu, M. Krikun, Y. Zhou, A. W. Yu, O. Firat, *et al.*, "Glam: Efficient scaling of language models with mixture-of-experts," in *International Conference on Machine Learning*, PMLR, 2022, pp. 5547–5569.

[37] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, *et al.*, "Palm: Scaling language modeling with pathways," *arXiv preprint arXiv:2204.02311*, 2022.

[38] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," in *ICLR*, 2016.

[39] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[40] M. Nagel, M. v. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1325–1334.

[41] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-Aware Automated Quantization with Mixed Precision," in *CVPR*, 2019.

[42] J. Lin, W.-M. Chen, Y. Lin, C. Gan, S. Han, *et al.*, "Mcunet: Tiny deep learning on iot devices," *Advances in Neural Information Processing Systems*, vol. 33, pp. 11 711–11 722, 2020.

[43] S. Shen, Z. Dong, J. Ye, L. Ma, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Q-bert: Hessian based ultra low precision quantization of bert," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 8815–8821.

[44] S. Kim, A. Gholami, Z. Yao, M. W. Mahoney, and K. Keutzer, “I-bert: Integer-only bert quantization,” in *International conference on machine learning*, PMLR, 2021, pp. 5506–5518.

[45] Z. Liu, Y. Wang, K. Han, W. Zhang, S. Ma, and W. Gao, “Post-training quantization for vision transformer,” *Advances in Neural Information Processing Systems*, vol. 34, pp. 28 092–28 103, 2021.

[46] H. Wang, Z. Zhang, and S. Han, “Spatten: Efficient sparse attention architecture with cascade token and head pruning,” *CoRR*, vol. abs/2012.09852, 2020. arXiv: 2012.09852. URL: https://arxiv.org/abs/2012.09852.

[47] Y. Bondarenko, M. Nagel, and T. Blankevoort, “Understanding and overcoming the challenges of efficient transformer quantization,” in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 7947–7969. URL: https://aclanthology.org/2021.emnlp-main.627.

[48] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, “Improving neural network quantization without retraining using outlier channel splitting,” in *International conference on machine learning*, PMLR, 2019, pp. 7543–7552.

[49] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, “Gptq: Accurate post-training quantization for generative pre-trained transformers,” *arXiv preprint arXiv:2210.17323*, 2022.

[50] G. Park, B. Park, S. J. Kwon, B. Kim, Y. Lee, and D. Lee, “Nuqmm: Quantized matmul for efficient inference of large-scale generative language models,” *arXiv preprint arXiv:2206.09557*, 2022.

[51] X. Wei, Y. Zhang, X. Zhang, R. Gong, S. Zhang, Q. Zhang, F. Yu, and X. Liu, *Outlier suppression: Pushing the limit of low-bit transformer language models*, 2022. DOI: 10.48550/ARXIV.2209.13325. URL: https://arxiv.org/abs/2209.13325.

[52] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, “Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension,” *arXiv preprint arXiv:1910.13461*, 2019.

[53] J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, “Roformer: Enhanced transformer with rotary position embedding,” *arXiv preprint arXiv:2104.09864*, 2021.

[54] M. Zaheer, G. Guruganesh, K. A. Dubey, *et al.*, “Big bird: Transformers for longer sequences,” in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020*, H. Larochelle, M. Ranzato, R. Hadsell, M.-F. Balcan, and H.-T. Lin, Eds., Virtual: Curran Associates, Inc., 2020.

[55] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linformer: Self-attention with linear complexity,” 2020. arXiv: 2006.04768 `[cs.CL]`.

[56] N. Kitaev, L. Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” in *8th International Conference on Learning Representations, ICLR 2020*, OpenReview.net, Addis Ababa, Ethiopia, Apr. 2020.

[57] bloc97, *NTK-Aware Scaled RoPE allows LLaMA models to have extended (8k+) context size without any fine-tuning and minimal perplexity degradation.* 2023. URL: https://www.reddit.com/r/LocalLLaMA/comments/14lz7j5/ntkaware_scaled_rope_allows_llama_models_to_have/.

[58] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, *Lost in the middle: How language models use long contexts*, 2023. arXiv: 2307.03172 [cs.CL].

[59] D. Li, R. Shao, A. Xie, Y. Sheng, L. Zheng, J. E. Gonzalez, I. Stoica, X. Ma, and H. Zhang, *How long can open-source llms truly promise on context length?* 2023. URL: https://lmsys.org/blog/2023-06-29-longchat.

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

[61] D. Paperno, G. Kruszewski, A. Lazaridou, N. Q. Pham, R. Bernardi, S. Pezzelle, M. Baroni, G. Boleda, and R. Fernández, “The LAMBADA dataset: Word prediction requiring a broad discourse context,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1525–1534. DOI: 10.18653/v1/P16-1144. URL: https://aclanthology.org/P16-1144.

[62] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, “Hellaswag: Can a machine really finish your sentence?” *CoRR*, vol. abs/1905.07830, 2019. arXiv: 1905.07830. URL: http://arxiv.org/abs/1905.07830.

[63] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, “Piqa: Reasoning about physical commonsense in natural language,” in *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[64] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, “Winogrande: An adversarial winograd schema challenge at scale,” *arXiv preprint arXiv:1907.10641*, 2019.

[65] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal, “Can a suit of armor conduct electricity? a new dataset for open book question answering,” in *EMNLP*, 2018.

[66] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, “GLUE: A multi-task benchmark and analysis platform for natural language understanding,” *CoRR*, vol. abs/1804.07461, 2018. arXiv: 1804.07461. URL: http://arxiv.org/abs/1804.07461.

[67] M. Roemmele, C. A. Bejan, and A. S. Gordon, “Choice of plausible alternatives: An evaluation of commonsense causal reasoning,” in *Logical Formalizations of Commonsense Reasoning, Papers from the 2011 AAAI Spring Symposium, Technical Report SS-11-06, Stanford, California, USA, March 21-23, 2011*, AAAI, 2011. URL: http://www.aaai.org/ocs/index.php/SSS/SSS11/paper/view/2418.

[68] S. Merity, C. Xiong, J. Bradbury, and R. Socher, *Pointer sentinel mixture models*, 2016. arXiv: 1609.07843 [cs.CL].

[69] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, "Measuring massive multitask language understanding," *CoRR*, vol. abs/2009.03300, 2020. arXiv: 2009.03300. URL: https://arxiv.org/abs/2009.03300.

[70] A. Williams, N. Nangia, and S. Bowman, "A broad-coverage challenge corpus for sentence understanding through inference," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, New Orleans, Louisiana: Association for Computational Linguistics, 2018, pp. 1112–1122. URL: http://aclweb.org/anthology/N18-1101.

[71] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[72] E. Almazrouei, H. Alobeidli, A. Alshamsi, *et al.*, *The falcon series of open language models*, 2023. arXiv: 2311.16867 [cs.CL].

[73] A. Q. Jiang, A. Sablayrolles, A. Mensch, *et al.*, *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL].

[74] A. Q. Jiang, A. Sablayrolles, A. Roux, *et al.*, *Mixtral of experts*, 2024. arXiv: 2401.04088 [cs.LG].

[75] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," *CoRR*, vol. abs/1909.08053, 2019. arXiv: 1909.08053. URL: http://arxiv.org/abs/1909.08053.

[76] Y. Bondarenko, M. Nagel, and T. Blankevoort, *Quantizable transformers: Removing outliers by helping attention heads do nothing*, 2023. arXiv: 2306.12929 [cs.LG].

[77] E. Miller, *Attention is off by one*, 2023. URL: https://www.evanmiller.org/attention-is-off-by-one.html.

[78] M. N. Team. "Introducing mpt-7b: A new standard for open-source, commercially usable llms." Accessed: 2023-05-05. (2023), URL: www.mosaicml.com/blog/mpt-7b (visited on 05/05/2023).

[79] S. Biderman, H. Schoelkopf, Q. Anthony, *et al.*, *Pythia: A suite for analyzing large language models across training and scaling*, 2023. arXiv: 2304.01373 [cs.CL].

[80] E. Almazrouei, H. Alobeidli, A. Alshamsi, *et al.*, "Falcon-40B: An open large language model with state-of-the-art performance," 2023.

[81] J. W. Rae, A. Potapenko, S. M. Jayakumar, C. Hillier, and T. P. Lillicrap, "Compressive transformers for long-range sequence modelling," in *International Conference on Learning Representations*, 2020.

[82] L. Gao, S. Biderman, S. Black, *et al.*, "The Pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[83] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, "Think you have solved question answering? try arc, the ai2 reasoning challenge," *arXiv:1803.05457v1*, 2018.

[84] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, "Piqa: Reasoning about physical commonsense in natural language," in *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[85] Together, *Llama-2-7b-32k-instruct — and fine-tuning for llama-2 models with together api*, 2023. URL: https://together.ai/blog/llama-2-7b-32k-instruct.

[86] T. Wolf, L. Debut, V. Sanh, *et al.*, *Huggingface's transformers: State-of-the-art natural language processing*, 2020. arXiv: 1910.03771 [cs.CL].

[87] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han, "Awq: Activation-aware weight quantization for llm compression and acceleration," in *MLSys*, 2024.

[88] W. Shao, M. Chen, Z. Zhang, P. Xu, L. Zhao, Z. Li, K. Z. Zhang, P. Gao, Y. Qiao, and P. Luo, "Omniquant: Omnidirectionally calibrated quantization for large language models," *arXiv preprint arXiv:2308.13137*, 2023.

[89] X. Wei, Y. Zhang, Y. Li, X. Zhang, R. Gong, J. Guo, and X. Liu, "Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling," *arXiv preprint arXiv:2304.09145*, 2023.

[90] T. Darcet, M. Oquab, J. Mairal, and P. Bojanowski, *Vision transformers need registers*, 2023. eprint: arXiv:2309.16588.

[91] L. Chen, H. Zhao, T. Liu, S. Bai, J. Lin, C. Zhou, and B. Chang, *An image is worth 1/2 tokens after layer 2: Plug-and-play inference acceleration for large vision-language models*, 2024. arXiv: 2403.06764 [cs.CV].

[92] H. Dong, X. Yang, Z. Zhang, Z. Wang, Y. Chi, and B. Chen, *Get more with less: Synthesizing recurrence with kv cache compression for efficient llm inference*, 2024. arXiv: 2402.09398 [cs.LG].

[93] J.-H. Kim, J. Yeom, S. Yun, and H. O. Song, *Compressed context memory for online language model interaction*, 2024. arXiv: 2312.03414 [cs.LG].

[94] R. Liu, H. Bai, H. Lin, Y. Li, H. Gao, Z. Xu, L. Hou, J. Yao, and C. Yuan, *Intactkv: Improving large language model quantization by keeping pivot tokens intact*, 2024. arXiv: 2403.01241 [cs.CL].

[95] M. Sun, X. Chen, J. Z. Kolter, and Z. Liu, *Massive activations in large language models*, 2024. arXiv: 2402.17762 [cs.CL].

[96] N. Cancedda, *Spectral filters, dark signals, and attention sinks*, 2024. arXiv: 2402.09221 [cs.AI].