LLMFolder: Revisiting Constant Folding in Large Language Models

Gansen Hu
Shanghai Jiao Tong University
Institute of Parallel and Distributed Systems, School of Computer Science
hugansen@sjtu.edu.cn

Zhaoguo Wang*
Shanghai Jiao Tong University
Institute of Parallel and Distributed Systems, School of Computer Science
zhaoguowang@sjtu.edu.cn

Wei Huang
Shanghai Jiao Tong University
Institute of Parallel and Distributed Systems, School of Computer Science
boogiepop1230@sjtu.edu.cn

Jinglin Wei
Shanghai Jiao Tong University
Institute of Parallel and Distributed Systems, School of Computer Science
weijinglin@sjtu.edu.cn

Haibo Chen
Shanghai Jiao Tong University
Institute of Parallel and Distributed Systems, School of Computer Science
haibochen@sjtu.edu.cn

Abstract

Large language models (LLMs) demonstrate remarkable capabilities but face deployment challenges due to their massive parameter counts. While pruning can reduce model size, it leads to significant accuracy degradation under high compression ratios. We present a novel perspective inspired by constant folding in compiler optimization. Our approach enables parameter reduction by treating activation functions in LLMs as linear functions.

However, recent LLMs use complex non-linear activations like GELU that prevent direct application of this technique. We propose LLMFolder, which enables optimization of LLMs with non-linear activations by *partially* approximating them with linear functions in frequently occurring input ranges. For outlier inputs, LLMFolder employs an online predictor to dynamically fall back to original computations.

Our experiments demonstrate that LLMFolder achieves 80% parameter reduction in feed-forward networks, while significantly outperforming state-of-the-art pruning methods with up to 65% higher accuracy. When combined with quantization and pruning, LLMFolder can further achieve 92.5% parameter reduction with only 4.4% average accuracy drop for a 7B model, where neither technique alone or combination can achieve. In practical deployments for a 7B model, LLMFolder achieves 1.6× end-to-end inference speedup when integrated with the vLLM serving system, and 1.4× speedup with the widely adopted HuggingFace implementation, while incurring only a 10.9% accuracy trade-off.

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Software and its engineering** → *Compilers.*

Keywords: large language model serving, constant folding

ACM Reference Format:
Gansen Hu, Zhaoguo Wang, Wei Huang, Jinglin Wei, and Haibo Chen. 2026. LLMFolder: Revisiting Constant Folding in Large Language Models. In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3767295.3769339

1 Introduction

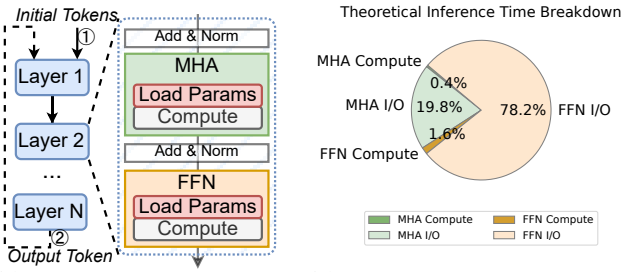
Transformer-based large language models (LLMs) demonstrate remarkable capabilities in natural language processing, but their deployment is challenged by vast parameter counts (billions to trillions) [18, 69, 71, 81]. Compression techniques like pruning [52, 53, 63, 64, 78], quantization [33, 51, 74], knowledge distillation [38, 66], and low-rank approximation [23, 49, 61] aim to mitigate this. However, pruning, though simple and popular [73], often incurs significant accuracy loss at high compression ratios (e.g., 80% compression can cause 70% accuracy loss).

Inspired by the success of constant folding [1] in traditional compiler optimizations, we propose a novel approach to LLM compression that maintains high accuracy even under aggressive compression ratios. This approach is orthogonal to existing techniques and can be combined with them for further efficiency gains. Constant folding, a fundamental compiler optimization technique that reduces both program code size and runtime computation costs, has been successfully employed across numerous programming languages[4, 5, 9]. We discover a unique opportunity to adapt this technique for LLM compression.

*Corresponding author (zhaoguowang@sjtu.edu.cn)



This work is licensed under a Creative Commons Attribution 4.0 International License.
EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/26/04
https://doi.org/10.1145/3767295.3769339



(a) The inference process of a transformer based LLM

(b) Theoretical Breakdown of the inference time on an RTX 4090

Figure 1. The inference process of transformer based LLM (a), where initial tokens are user prompts, output tokens are response from LLM; the inference time breakdown (b) with 91 initial tokens, 178 output tokens for Falcon-7B (Other operations takes <0.5% time and not shown).

The architecture of an LLM primarily consists of two core components: the Multi-Head Attention (MHA) block and the feed-forward network (FFN) block, with the FFN block accounting for 67% to 80% [19] of the total parameters. The FFN block comprises two matrix multiplications separated by a non-linear activation function (e.g., ReLU or GELU).

Our key observation is that through linear approximation of the non-linear activation function, we can leverage the associative property of matrix multiplication to merge the two FFN matrices into a single, compact matrix. This transformation enables a reduction in FFN parameters by up to 87.5% in theory (detailed in Section 3.1).

However, modern LLMs predominantly use sophisticated non-linear activation functions, particularly GELU [41] (adopted in GPT series models [21]) and SiLU [32] (used in LLaMA2/3 [37, 68]). These complex functions are crucial for capturing intricate patterns in the data, and naively approximating them with linear functions leads to severe performance drop.

In this paper, we present LLMFolder, a novel approach that enables constant folding in LLMs with non-linear activation functions at low approximation error, and is orthogonal to existing compression approaches. Our key insight is that the inputs to these activation functions in LLMs are typically concentrated within a narrow range. This distribution pattern allows us to *partially* approximate the non-linear activation function with a linear function in the "hot" input ranges (where most values occur), introducing minimal approximation error. For outlier inputs that fall outside these ranges, LLMFolder employs an efficient online predictor to identify them and dynamically falls back to the original activation function computation. Experiments demonstrate that LLMFolder reduces the FFN weight matrices by up to 80%, while outperforming state-of-the-art pruning methods in accuracy by up to 65%, and compatable with existing quantization methods. When combined with quantization (GPTQ) and pruning (Wanda), LLMFolder achieves 92.5% parameter reduction with only 4.4% accuracy drop for a 7B model, where neither alone or combination can achieve. For inference, LLMFolder delivers up to 1.6× end-to-end speedup on the state-of-the-art vLLM serving system [45], and 1.4× speedup on HuggingFace's widely-used implementation [10].

The key contributions of this paper are:

- We identify a novel opportunity to apply constant folding to LLMs. Our method treats activation functions as partially linear functions, reducing FFN block parameters by up to 87.5% in theory.
- We introduce LLMFolder, a new system for constant folding in LLMs. It handles non-linear activation functions through partial linear approximation and includes efficient fallback mechanisms.
- We conduct extensive experiments with LLMFolder. Results show up to 80% actual parameter reduction and could achieve better accuracy-compression ratio tradeoff when combined with pruning or quantization.

2 Background and Motivation

In this section, we first cover the basics of LLM inference and highlight the performance bottlenecks introduced by massive parameters. We then review existing approaches to reducing LLM size and identify their constraints.

2.1 Large Language Model Inference

LLM Inference Overview. LLM inference generates output tokens in response to input tokens through two main phases. In the first phase, as shown in Figure 1a, the model processes all initial input tokens through its N layers sequentially to establish context. In the next phase, the model generates output tokens one at a time in an auto-regressive manner, where each generated token becomes part of the input sequence for predicting the next token. This process continues until either the desired output length is reached or a stop condition is met.

Operations in LLM Inference. The transformer architecture, which forms the backbone of LLMs, consists of two main components: the multi-head attention (MHA) block and the feed-forward network (FFN) block. The MHA block enables the model to focus on different parts of the input sequence simultaneously through multiple attention heads, while the FFN block, composed of two linear transformations with an activation function in between, processes the features from the attention block. As Figure 1a shows, an LLM typically stacks multiple transformer layers, where each layer sequentially processes the input through its MHA and FFN blocks. During inference, the model parameters for both blocks need to be loaded from memory before performing computations on input tokens.

2.2 Cost of LLM Inference

Despite their remarkable capabilities, deploying LLMs for inference tasks remains challenging due to their massive

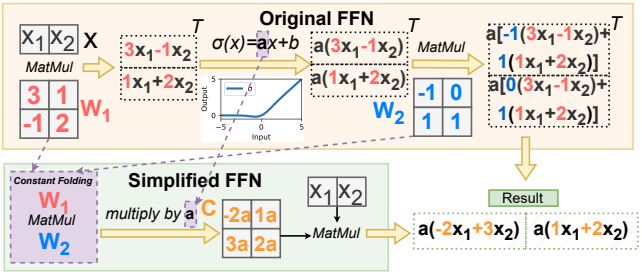


Figure 2. Illustration of constant folding in the FFN block of an LLM. The FFN contains two weight matrices W_1, W_2. By treating activation function σ as a linear function $ax + b$, matrix multiplications can be reordered from $a(xW_1)W_2$ to $x(aW_1W_2)$. Since aW_1W_2 consists of only constants, it can be pre-computed into a single matrix C before inference.

parameter counts [51, 52, 69, 71]. This large parameter count significantly impacts inference performance.

To quantify this impact, we analyzed the inference time breakdown of Falcon-7B using the SharedGPT dataset [14], which contains 90K real-world chat histories between humans and LLMs. 80% of the total parameters are in the FFN blocks of Falcon-7B. Using the dataset's average lengths (91 tokens for human input, 178 tokens for LLM response), we measured the inference time with the vLLM inference engine [7] on an RTX 4090, where attention blocks take 0.9s, and the FFN blocks takes 2.1s, showing the bottleneck is in FFN blocks. We also performed a therotical analysis of the time spent on computation and I/O operations (performed on the 1TB/s bandwidth VRAM of RTX 4090) in both MHA and FFN blocks. Figure 1b shows that parameter loading I/O dominates the inference time, with the FFN I/O alone consuming 78.2% of the total time. This high I/O overhead stems from LLMs' auto-regressive nature: generating each token requires loading all model parameters into the hardware's cache (Figure 1a), resulting in repeated, expensive I/O operations [69, 71, 81].

2.3 LLM Compression

To alleviate the I/O overhead in LLM inference, various compression techniques have emerged. These are broadly categorized into pruning [52, 53, 63, 64, 78], quantization [33, 43, 51, 70], knowledge distillation [38, 66], and low-rank approximation [23, 49, 61]. Such techniques primarily aim to reduce the memory required for storing model parameters [33, 38, 49, 64] and/or the bandwidth needed for loading them into hardware cache [52, 63].

One-shot pruning and post-training quantization are prominent LLM compression techniques, as they obviate the need for expensive retraining [73]. Pruning identifies and removes less critical parameters (e.g., based on magnitude or gradient), reducing both GPU memory footprint and the I/O cost of loading parameters from memory to hardware cache. Quantization reduces the bit-width of model parameters, similarly

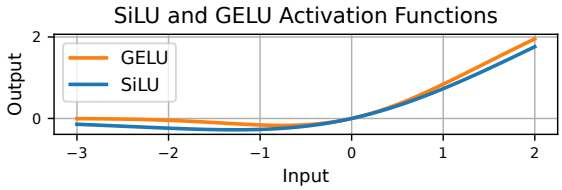


Figure 3. Figure of SiLU and GELU activation functions in a input range of `[-3, 2]`.

decreasing the I/O cost for parameter loading from memory. These techniques can be combined for enhanced compression efficacy [35, 39, 52, 63, 80].

However, pruning can severely degrade model accuracy at high compression ratios. We demonstrate by evaluating state-of-the-art pruning methods—unstructured techniques Wanda [64] and RIA [78], and contextual method PowerInfer [63]—on Falcon-7B across three benchmarks [12, 16, 20] (detailed in Section 7.2). While a 50% compression ratio (for unstructured pruning) leads to a modest accuracy decline of 0.9%-4.8%, an 80% compression can reduce accuracy to zero on the Lambada benchmark, rendering the model ineffective. For contextual pruning (PowerInfer), performance degradation is even more pronounced, due to non-ReLU-based LLMs' low activation sparsity [56]. Such severe accuracy drops can also hinder the combination of pruning with other methods.

3 Opportunity and Challenge

Inspired by compiler optimizations, we propose a novel approach to reduce I/O costs in LLMs: simplifying FFN blocks by modifying activation functions. This method reduces parameters through constant folding, achieving higher accuracy under aggressive compression than pruning.

3.1 Opportunity

A key insight for simplifying FFN blocks in LLMs is that their activation functions can be approximated as linear functions. The FFN block in transformer models consists of two weight matrices W_1, W_2, and an activation function between them. As shown in Figure 2, a simple FFN block with activation computes:

$$FFN(x) = \sigma(xW_1)W_2$$

where matrix W_1 is of size $d \times h$ and W_2 is $h \times d$, σ is the activation function. A function $f(x)$ is linear if and only if it can be expressed as $f(x) = ax + b$, where a and b are constants. When the activation function is approximated by a linear function, the FFN becomes the following linear transformation:

$$FFN(x) = a((xW_1) + b^1)W_2$$

This linear representation of activation function enables us to leverage matrix computation properties to optimize the FFN block. Specifically, we can reorder the computation

[1]To ease presentation, we omit b by regarding it as 0, without lossing generalizability.

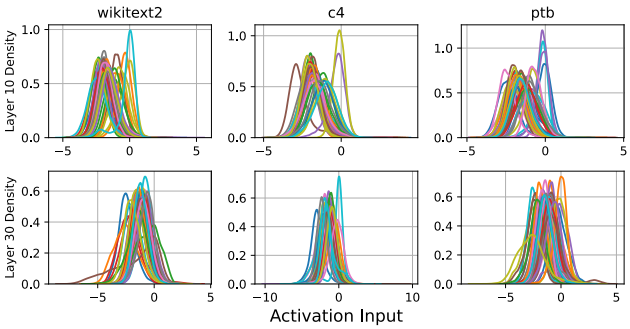


Figure 4. Density estimation of the activation function input distribution for 50 neurons of layer 10 and 30 in Falcon-7B. Each curve represents the kernel density estimation of the input distribution for a single neuron. The higher the density, the more activation inputs fall into the range. The left two figures are profiled on WikiText2, the middle two on C4, and the right two on PTB.

from $a(xW_1)W_2$ to $x(aW_1W_2)$ using matrix associativity. This reordering would not be possible with non-linear activation functions. The key advantage is that aW_1W_2 contains only constants and can be pre-computed into a single matrix C before inference. As illustrated in Figure 2, with input $[x_1, x_2]$, the FFN computes $\texttt{t}=[a(-2x_1 + 3x_2), a(x_1 + 2x_2)]$. By pre-computing aW_1W_2 into matrix $C = \begin{pmatrix} -2a & a \\ 3a & 2a \end{pmatrix}$, we can simplify the entire FFN into a single matrix multiplication xC that produces the same result $\texttt{t}$. This transformation significantly reduces the parameter count in FFN blocks. Originally, an FFN block with weight matrices $W_1 \in \mathbb{R}^{d\times h}$ and $W_2 \in \mathbb{R}^{h\times d}$ contains $2 \times d \times h$ parameters. After folding into a single matrix $C \in \mathbb{R}^{d\times d}$, it contains only d^2 parameters. For typical LLM architectures like GPT2 [3], BLOOM [46], and Falcon [19] where $h = 4d$, this reduces the parameter count by up to 87.5% per FFN block.

3.2 Challenge

While constant folding in LLMs shows promise, a significant challenge arises from the widespread adoption of sophisticated non-linear activation functions in modern language models [25, 29, 32, 41, 44]. Our analysis of the top 15 text generation models on HuggingFace [10], a leading model repository, reveals that all models use complex activations - 7 employ GELU and 8 use SiLU. These sophisticated activation functions make naive linear approximation challenging.

Consider GELU as an example. As illustrated in Figure 3, GELU exhibits a smooth but intricate input-output relationship. When we attempt to approximate GELU with a simple linear function (as described in Section 3.1), the model suffers substantial accuracy degradation. To validate this, we modified Falcon-7B by replacing its GELU activation with a linear function in the FFN blocks, derived through linear regression on the original activation's input-output data. When evaluated on the Lambada [12] benchmark, the model's accuracy plummeted from 75% to 0, making it impractical for real-world applications.

Model Name	Activation Function	Wikitext-2	C4	PTB
Falcon-7B	GELU	20.2%	20.6%	20.7%
Falcon-40B	GELU	19.6%	19.7%	20.1%
BLOOMZ-7B1	GELU	19.1%	19.4%	19.8%
LLaMA2-7B	SiLU	18.4%	18.5%	18.2%

Table 1. Average percentage of activation input ranges containing 65% of inputs relative to total range length, measured across neurons in three datasets.

4 Insights and Overview

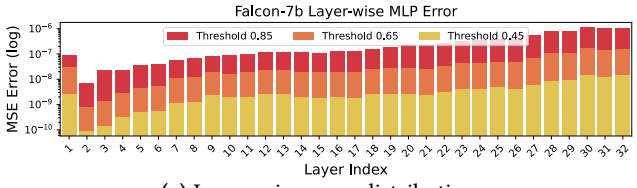
Through careful analysis of LLM internals, we have identified two key insights that enable effective approximation of non-linear activation functions with linear ones while preserving model accuracy.

4.1 Insights

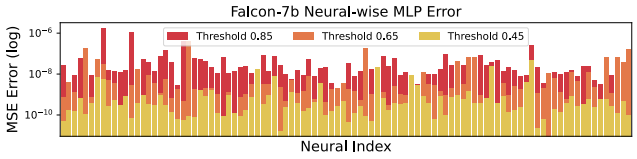
Insight 1. Skewed distribution of activation function inputs. The input distribution for activation functions in LLMs exhibits a skewed pattern, with inputs predominantly concentrated within a narrow range of the input space. To validate this observation, we analyzed three language task datasets: Wikitext-2 [55], C4 [60], and PTB [54]. A FFN block consists of multiple *neurons*, where each neuron processes input through its weights and activation function. The activation input distribution at the neuron level is crucial for understanding model behavior [52, 63]. We examined this distribution within FFN weight matrices. Each weight matrix can be decomposed into individual neurons, which represent distinct computational units. For example, in Figure 2, W_1 comprises two neurons represented as columns: $\begin{pmatrix} 3 \\ -1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$. Using 20K random tokens from each dataset, we collected comprehensive statistics.

Figure 4 illustrates the input activation distribution for 50 randomly sampled neurons from two layers of Falcon-7B across these datasets. Our analysis reveals consistent distribution patterns within the same layer across different datasets. Notably, approximately 65% of activation inputs for each neuron are concentrated within just 20% of the total input range, demonstrating significant skewness. This pattern holds true across various model architectures. Table 1 presents the average range portion containing 65% of activation inputs relative to the total range for four LLMs: Falcon-7B/40B, Bloomz-7B1, and LLaMA2-7B. While the first three models employ GELU activation, LLaMA2-7B uses SiLU. Across all models and datasets, this portion consistently remains between 18-20%. These findings align with previous research [52, 57, 63], confirming that skewed activation input distribution might be an inherent characteristic of LLMs. This insight reveals that activation function inputs follow a power law-like distribution, enabling effective approximation of non-linear activation functions within a compact "hot" input range.

Insight 2. Non-uniform distribution of layer and neuron importance. Not all layers and neurons contribute



(a) Layer-wise error distribution



(b) Neuron-wise error distribution

Figure 5. Error distribution of layer, and neuron in the 10-th layer in Falcon-7B. The error is profiled on the C4 dataset.

equally to the model's performance. Using the same methodology as in the previous insight, we collected the Mean Square Error (MSE) of approximating the activation function with linear functions for each neuron in each layer. Specifically, for each layer and neuron, we first identify the input range containing a certain portion (e.g., 85%) of activation inputs based on profiled distribution. We then approximate the activation function within this range by linear function $y = ax + b$, where parameters a and b are obtained through linear regression to minimize the loss within the range. This process is repeated for different portions of inputs (from 65% to 95%) to analyze the approximation error patterns.

Figure 5.a shows significant differences in errors across layers. For instance, when 85% of inputs fall within the linear range, the error of the first layer (10^{-7}) is an order of magnitude larger than that of the second layer (10^{-8}). Previous works have also shown that layer importance is non-uniformly distributed [75, 76, 79]. Similarly, neuron importance within a layer varies significantly. Figure 5.b presents the error distribution for 1% of neurons randomly selected from the first layer of Falcon-7B, showing error variations by nearly three orders of magnitude (from 10^{-10} to 10^{-7}). Previous studies have also found similar patterns, where neuron activation is non-uniformly distributed, with outliers having a larger impact on model performance [30, 51, 63]. In summary, Figure 5 highlights the non-uniform distribution of layer and neuron importance, underscoring the need to treat each layer and neuron distinctly in our design approach.

4.2 Overview

Armed with these insights, we propose LLMFolder, the first system that enables constant folding in LLMs while preserving model capabilities and reducing parameters. Our approach leverages two observations: 1) the skewed distribution of activation inputs allows us to focus approximation on the most frequently accessed input ranges; 2) the varying importance of different layers and neurons suggests that approximation thresholds should be adapted accordingly.

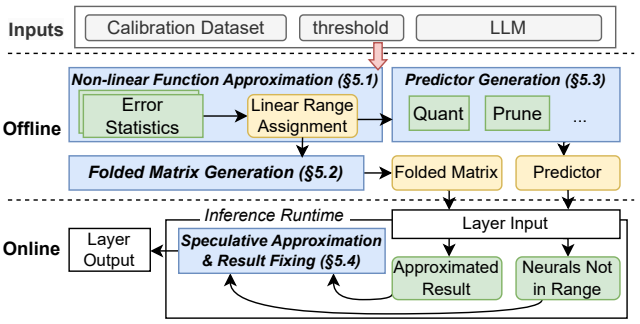


Figure 6. Architecture of LLMFolder with offline and online components. The offline component generates the constant folded matrix and predictor based on the calibration dataset. The online component performs inference using the constant folded matrix and predictor.

As illustrated in Figure 6, LLMFolder takes three inputs: an LLM, a calibration dataset, and a threshold t that defines what portion of inputs should fall within the linear approximated range. The system generates two key outputs: a compressed model weight matrix (produced via constant folding based on approximated linear ranges) and a predictor that identifies neurons operating outside their approximated ranges.

The system architecture consists of offline and online components. The offline component performs two tasks. First, it analyzes the calibration dataset–a small number of text samples–to collect: 1) activation input distributions for each layer and neuron, and 2) importance scores based on approximation errors when replacing activation functions with linear functions. Based on these statistics, LLMFolder approximates the activation function with linear functions, focusing on the "hot" input ranges where most activations occur. The approximation range for each neuron is determined by its importance score, which LLMFolder calculates based on the error introduced by linear approximation under different thresholds. The final outputs are the constant folded matrix and a predictor for identifying out-of-range neuron activations. During inference, the online component first performs a speculative computation using the constant folded matrix. It then uses the predictor to identify neurons operating outside their approximated ranges. For these identified neurons, it corrects the results by: 1) subtracting the incorrectly computed linear approximations, and 2) recomputing the actual results using the original activation function.

Figure 7 illustrates this process using a simple FFN with two neurons (each neuron corresponds to one column in W_1 and one row in W_2, as shown in yellow and green). The FFN uses GELU as activation function. In the offline phase, LLMFolder determines linear approximations for each neuron's frequent input range. For neuron 1, the range is $(-1.5, 0.12)$ with slope 0.25 and intercept 0.1; for neuron two, the range is $(-3.5, -0.1)$ with slope 0.1 and intercept 0.2. These ranges are selected based on calibration data to ensure at least t

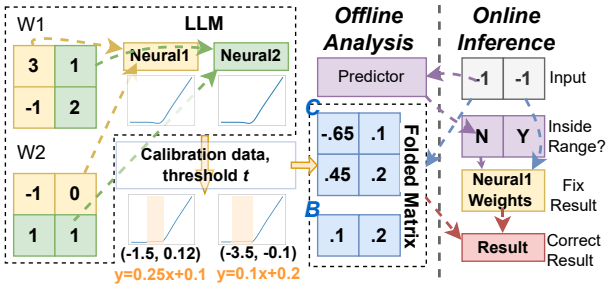


Figure 7. Example of an FFN in an LLM constant folded by LLMFolder. The constant folded matrix is generated based on approximated linear ranges for each neuron. The predictor identifies neuron one is not in the approximated range. The online component fixes the FFN result by subtracting the wrong result and recomputing the actual result using the original activation function.

proportion of inputs fall within them. The system then generates constant folded matrices C and B using these linear parameters. When inference with input token $x = \begin{pmatrix} -1 & -1 \end{pmatrix}$, LLMFolder 1) performs speculative computation: $xC + B = \begin{pmatrix} 0.3 & -0.1 \end{pmatrix}$; 2) uses predictor to determine that only neuron two's input falls within its approximated range; 3) corrects the result by a) subtracting neuron 1's approximate result: $(0.25x\begin{pmatrix} 3 \\ -1 \end{pmatrix} + 0.1)\begin{pmatrix} -1 & 0 \end{pmatrix} = \begin{pmatrix} 0.4 & 0 \end{pmatrix}$, and b) adding back neuron 1's actual result: $GELU(x\begin{pmatrix} 3 \\ -1 \end{pmatrix})\begin{pmatrix} -1 & 0 \end{pmatrix}$. This correction mechanism ensures accurate results while preserving the efficiency benefits of constant folding where applicable.

5 LLMFolder Design

In this section, we present the detailed design of LLMFolder, which comprises offline and online components. The offline component consists of three key aspects: approximating non-linear activation functions with linear functions (Section 5.1), generating constant folded matrices (Section 5.2), and constructing predictors (Section 5.3). The online component details how these elements work together during inference (Section 5.4).

5.1 Non-linear Function Approximation

LLMFolder aims to reduce FFN parameter counts and speedup inference by replacing complex non-linear activation functions with simple linear functions where possible. At a high level, our non-linear activation function replacing approach works as follows. For each FFN layer that consists of two weight matrices (W_1 and W_2) and a non-linear activation function σ in between, we 1) break down the computation of the entire FFN into computations on individual neurons. 2) For each neuron, analyze its input patterns to the activation function using a calibration dataset. 3) Replace the neuron's non-linear activation with linear functions ($y = ax + b$) within its most frequently observed input range. The linear functions are obtained using least squares regression.

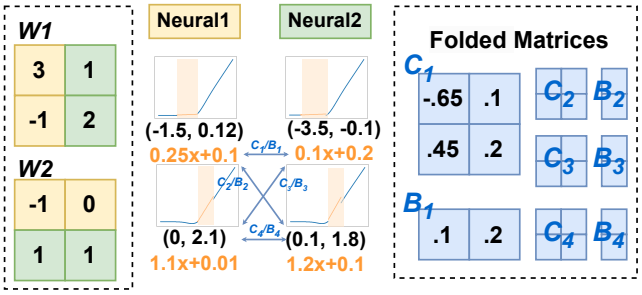


Figure 8. Example of the exponential growth in the number of folded matrices.

An intuitive design would be to replace each neuron's non-linear activation with multiple linear ranges, as different input ranges may exhibit distinct patterns. For example, for input ranges $[l_1, a)$, $[a, b)$, and $[b, l_2)$, we could fit three separate linear functions to better approximate the activation function. However, a significant challenge arises with this approach. When folding the weight matrices, we need a separate folded matrix for each combination of ranges across neurons. With h neurons and r ranges per neuron, this leads to r^h folded matrices. Figure 8 illustrates this exponential growth. With just 2 neurons and 2 ranges each, we need 4 diffesrent folded matrices to cover all possible combinations.

Given that modern language models typically have over 10,000 neurons per FFN layer, storing multiple ranges quickly becomes impractical. This leads us to adopt a single-range strategy, which we describe in detail in the following section.

5.1.1 Single Range Approximation Strategy

To overcome the exponential storage overhead of multi-range approximation, LLMFolder adopts a pragmatic single-range strategy. For each neuron, we approximate its non-linear activation using one linear function within a carefully chosen input range.

More specifically, for each neuron n, we aim to find a linear function $y = ax + b$ that best approximates the activation within a range $[l_1, l_2)$. The approximation function ϕ_n is defined as:

$$\phi_n(x) = \begin{cases} ax + b, & l_1 \le x < l_2 \\ \sigma(x), & otherwise \end{cases}$$

To quantify the quality of our approximation, we define the error metric for the entire FFN layer as: $err(x) = \sum_n^h err_n(x)$ where each neuron's error contribution is measured using the L2 distance [67] between original and approximated outputs: $err_n(x) = ||\sigma(xW_{1,:,n})W_{2_{n,:}} - \phi_n(xW_{1,:,n})W_{2_{n,:}}||_2$.

Adaptive Thresholding. Given the definition of FFN approximation error, LLMFolder needs to determine appropriate linear approximation ranges for each neuron while satisfying the user-specified in-range threshold t. A key challenge is deciding how much of each neuron's input should be covered by the linear approximation.

Based on our insight in Section 4.1 that different layers and neurons contribute unequally to the model's performance,

we develop an adaptive thresholding mechanism that assigns different coverage requirements to different components. This mechanism operates at two levels:

1) *Layer-level thresholding*: For each layer i, we assign a threshold t_i that determines what percentage of inputs should be linearly approximated. This is formulated as an optimization problem: $minimize \sum_{i}^{L} E_i t_i,\ s.t., \sum_{i}^{L} t_i = tL$, where E_i represents the total empirical error measured on a calibration dataset for layer i, L is the total number of layers, and t is the target threshold. The empirical error E_i is estimated by approximating all neurons in layer i under threshold t_i (e.g., as Figure 5a shows). The constraint ensures that the average threshold across layers equals the target.

2) *Neuron-level thresholding*: Within each layer i, we further distribute the layer's threshold t_i across its neurons using: $minimize \sum_{n}^{h} E_{i_n} t_{i_n},\ s.t., \sum_{n}^{h} t_{i_n} = t_i h$, where E_{i_n} represents the error contribution of neuron n in layer i, h is the number of neurons per layer, and t_{i_n} is the threshold assigned to neuron n. This ensures more important neurons receive more conservative thresholds.

This two-level adaptive thresholding approach allows LLMFolder to maintain high approximation accuracy while satisfying user's threshold requirement. More critical layers and neurons are assigned stricter thresholds, while less important ones can use more aggressive linear approximations.

Range Searching. For each neuron, LLMFolder employs a greedy search strategy to determine the optimal linear approximation range, following these steps. 1) Initialize a starting range centered at the input distribution's centroid, computed using kernel density estimation (KDE) [59, 65]. 2) Gradually expand the range boundaries until reaching the neuron-specific coverage threshold t_{i_n}. 3) During expansion, select the direction (left/right) that produces minimal approximation error. 4) Once the optimal range is determined, fit a linear function using least squares regression. The complete algorithm is formalized in Algorithm 1, leveraging a small calibration data to ensure both approximation accuracy and computational efficiency without fine-tuning the model.

In the rare case that the determined linear approximation ranges are suboptimal (i.e., they cover few inputs), the primary impact is on inference latency, not accuracy. Latency increases because more inputs fall outside their linear range, requiring a fallback to the slower, original non-linear activation (detailed in Section 5.4). Accuracy remains largely unaffected because these out-of-range inputs are processed by the original, precise activation function, preserving the final output quality.

Algorithm 1: Non-linear Function Approximation

Input: The non-linear function σ, threshold t, large language model M, calibration dataset X, range increment step s.
Output: Set of generated linear ranges P for each neuron in each layer.
$err_{layers} \leftarrow$estimate_error_layers($M.FFNs$, X)
$t_{layers} \leftarrow$error_aware_threshold(err_{layers}, t)
$P \leftarrow \{\}$
for t_i *in* t_{layers} **do**
  $m \leftarrow M.FFNs[i]$, $N \leftarrow \{\}$
  $err_{neurons} \leftarrow$estimate_error_neurons(m, t_i, X)
  $t_{neurons} \leftarrow$error_aware_threshold($err_{neurons}$, t_i)
  for t_{i_n} *in* $t_{neurons}$ **do**
    /* kernel density estimation to find centroid */
    $centroid \leftarrow$find_centroid($m.neuron_n$, t_{i_n}, X)
    /* initialize the range around centroid */
    $R \leftarrow$(left=centroid, right=centroid)
    while *portion of input covered by* $R < t_{i_n}$ **do**
      $R_l \leftarrow$(R.left - s, R.left)
      $R_r \leftarrow$(R.left, R.right + s)
      /* approximate the range with a linear regressior, and evaluate the error */
      $err_{left} \leftarrow$approx_error($m.neuron_n$, R_l, X)
      $err_{right} \leftarrow$approx_error($m.neuron_n$, R_r, X)
      if $err_{left} < err_{right}$ **then**
        $R \leftarrow R_l$
      else
        $R \leftarrow R_r$
    $N[n] \leftarrow R$
  $P[i] = N$
return P

5.2 Constant Folded Matrix Generation

Given the approximated linear ranges for each neuron in each layer, LLMFolder generates constant folded matrices C and bias vectors B through a systematic transformation.

For each FFN layer, LLMFolder first substitutes the activation function with the approximated linear function ϕ_n for each neuron n. This transformation creates a sequence of three consecutive linear operations per neuron: an initial linear transform (on $W_{1_{:,n}}$), followed by the approximated linear transform (on ϕ_n in $[l_1, l_2)$), and finally a second linear transform (on $W_{2_{n,:}}$). These three transformations are then constant folded into a single linear transformation. The constant folding process for each neuron can be formally expressed as: $CF(W_{1_{:,n}}, W_{2_{n,:}}, \phi_n) = (aW_{1_{:,n}} W_{2_{n,:}}, bW_{2_{n,:}}) = (C_n, B_n)$. Here, CF represents the constant folding function for neuron n, producing a matrix $C_n \in \mathbb{R}^{d \times d}$ and a bias vector $B_n \in \mathbb{R}^{1 \times d}$. The parameters a and b denote the slope and intercept of

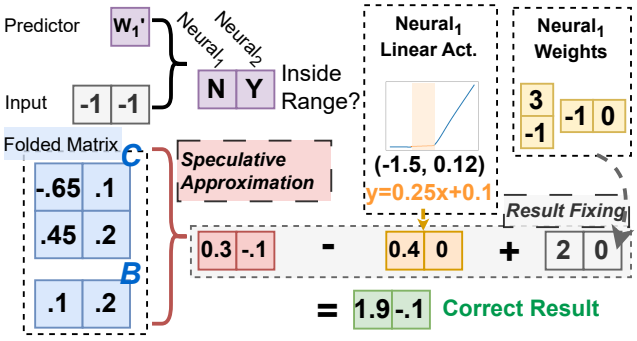


Figure 9. The speculative approximation and result fixing steps in the online phase of LLMFolder.

the approximated linear function ϕ_n, respectively. To complete the transformation for each layer, LLMFolder summarizes all h individual neuron's folded matrix into a single consolidated matrix $C \in \mathbb{R}^{d\times d}$ and bias vector $B \in \mathbb{R}^{1\times d}$: $(C, B) = (\sum_n^h C_n, \sum_n^h B_n)$ for each layer, respectively.

5.3 Predictor Generation

While Section 5.2 details optimized matrix generation for linear approximation, a practical mechanism is needed for inputs outside the approximated range (Section 4.1).

LLMFolder employs a *speculative approximation with result fixing* scheme. It optimistically assumes inputs are within the linear range, performs fast matrix operations, and then uses a lightweight predictor to identify and correct inaccurate results (Section 5.4).

The predictor identifies neurons with out-of-range inputs using a compressed version of the weight matrix W_1. This compressed matrix retains enough information for prediction without full matrix operation overhead. LLMFolder uses quantization [34] to compress W_1, balancing accuracy and simplicity over alternatives like smaller prediction networks [52, 63]. Other LLM compression methods (e.g., pruning, knowledge distillation) are viable if they preserve sufficient predictive information. This approach ensures high performance and accuracy through selective result fixing.

5.4 Inference Runtime

The LLMFolder operates in two phases: an offline phase that generates the constant folded matrix (Section 5.2) and predictor (Section 5.3), and an online phase that performs inference through speculative approximation and result fixing.

Speculative Approximation. During this step, LLMFolder approximates the FFN computation using a pre-computed constant folded matrix. This simplifies the complex FFN computation into a simple matrix multiplication and addition operation: $FFN(x) \approx xC + B$. This approximation significantly reduces the computational complexity compared to the original FFN computation.

Result Fixing. Not all neurons can be accurately approximated using the simplified computation above. The predictor identifies these cases, and LLMFolder corrects the results for these specific neurons using the original computation method. This correction involves two steps. First, we remove the approximate results for the identified neurons. Then, we compute and add back their actual results using the original FFN computation.

Figure 9 illustrates this process with a concrete example. The system first generates an approximate result, then identifies and corrects inaccurate approximations by replacing them with precise computations.

Memory Footprint. The memory footprint of LLMFolder consist of three main parts. 1) The constant folded matrix and its bias terms; 2) The lightweight predictor; 3) Only the original weights of neurons that require exact computation (fixing). Notice that although only a subset of original weights are needed to be loaded from memory into cache during inference, all the original weights are still retained in GPU memory for fast access. The system's memory efficiency varies between the two inference phases. In the first phase, more neurons might require their original weights, leading to higher memory usage. In contrast, during sequential token generation (second phase), fewer neurons typically need correction, resulting in better memory efficiency.

6 Implementation

We implemented LLMFolder as a prototype using PyTorch [6]. We integrated LLMFolder into the HuggingFace Transformer library (v4.41.0) [10] and vLLM inference engine (v0.6.6) [7]. The system consists of two main components:

Offline Component. The offline preprocessing pipeline, implemented in approximately 900 lines of Python code, handles non-linear function approximation and constant folding matrix generation. We leveraged cuML [11], a GPU-accelerated machine learning library, for training the linear regression model and performing kernel density estimation efficiently. The resulting constant-folded matrix is serialized to a binary format for runtime loading. The predictor is obtained by GPTQ [34] with 2-bit precision, which is also stored in binary format, and loaded before inference.

Runtime Component. The inference runtime consists of two parts: (1) a Python library implementing speculative approximation and result fixing logic (approximately 150 lines), and (2) a specialized CUDA kernel [2] for efficient result fixing. The kernel performs selective loading of FFN weights based on predicted neuron indices. We optimized the memory efficiency by memory coalescing [40] and vectorized shared memory access [13].

7 Evaluation

Our evaluation mainly focuses on the following questions:

Name	Params.	%Params. in FFN	Activation Function	#Downloads per month
Falcon-7B	7.2B	80%	GELU	75K
Falcon2-11B	11.1B	80%	GELU	19K
BLOOMZ-7B1	7.1B	67%	GELU	9.3K
GPT-2-XL	1.6B	67%	GELU	306K
OPT-6.7B	6.7B	67%	ReLU	39K

Table 2. Basic statistics of the evaluated LLMs. The third column shows the estimated percentage of total model parameters in the FFN layer.

- How does LLMFolder affect the model's performance in terms of perplexity and accuracy on downstream tasks?
- How does LLMFolder compare with existing LLM one-shot pruning and post-training quantization methods?
- How much pratical speedup can be gained during inference by LLMFolder?
- How LLMFolder performs when combined with quantization and pruning?

Apart from the above main questions, we evaluate the precision and effectiveness of the range assignment algorithm, predictor size's influence on model performance, runtime overhead, and the precision effects of reordering FFN.

7.1 Settings

Models. We evaluated LLMFolder on five large language models ranging from 1.6B to 11.1B parameters: Falcon-7B [19], Falcon2-11B [15], BLOOMZ-7B [46], GPT2-XL [3], and OPT-6.7B [17]. These models represent different architectures and training objectives while being widely adopted in real-world applications. Each model employs a standard FFN architecture with either GELU or ReLU activation functions. Table 2 summarizes the key characteristics of these models, including their parameter count, architecture details, and type of activation functions.

Evaluating Benchmarks. Following prior works evaluating LLM performance [33, 51, 53, 64], we evaluate models on language generation and zero-shot tasks with lm-evaluation-harness [36]. For language generation, we measure perplexity on Wikitext-2 [55], C4 [60], and Penn Treebank (PTB) [54]. On language generation tasks, perplexity measures how well a model predicts text by calculating the exponential of the average negative log-likelihood. Lower perplexity indicates better performance, with a perplexity of 1 representing perfect prediction. For zero-shot evaluation, we test accuracy on PIQA [20], Lambada [12], and ARC-Challenge [16], which cover reasoning, word prediction, and question answering tasks respectively. Higher accuracy indicates better model performance on these tasks.

Setup. We conducted all our experiments on a single NVIDIA RTX 4090D GPU with 24GB memory. Unless specified, the calibration dataset we used consists of 8 samples randomly choosen from the first shard of C4, each with 2048 tokens. C4 is a dataset consisting of web pages scraped from the internet, representing generic text data and making sure that we do not overfit to specifics tasks.

Baselines. We compare LLMFolder against five model compression methods of three categories: 1) Wanda [64] and RIA [78]: Pruning techniques that remove model parameters based on activation inputs and weights. 2) PowerInfer [63]: A contextual pruning method that dynamically skips computations by predicting activation sparsity patterns. 3) GPTQ [34] and AWQ [51]: Post-training quantization methods that reduce model weight precision to lower bit representations. In all experiments, we compress the FFN blocks using LLMFolder or the baseline methods while keeping the attention blocks unchanged. LLMFolder's compression ratio is controlled by the threshold of input activations falling within the linear range (Section 5.1). We also account for the additional storage required for the folded matrix and predictor, which together constitute approximately 10% and 4.8% of the model size, respectively (Section 5.4). The overall compression ratio is averaged across both prefill and decode phases.

For Wanda and RIA, the compression ratio is determined by the pruning ratio or the quantization bits, with pruned weights considered compressed. For PowerInfer, the compression ratio is controlled by the percentage of activated neurons in the FFN block, following the approach in [63] for managing sparsity. For GPTQ and AWQ, the compression ratio is based on the quantization bit-widths of 8, 4, 3, and 2 bits (with all original models in 16 bits). The additional storage required for quantization scales and zero-points is also included in the calculations.

7.2 Folded Model's Performance

We quantifies the performance of the models folded by LLMFolder on language generation and zero-shot tasks, then compare with the baselines for varying compression ratios.

Summary. LLMFolder outperforms Wanda, RIA and PowerInfer across different compression ratios, and achieves comparable result to GPTQ and AWQ.

1) At lower compression ratios (50% and 70%), LLMFolder demonstrates marginally better performance in both language generation and zero-shot tasks compared with Wanda and RIA achieving 4.6× and 3.1× lower perplexity, 1.47× and 1.26× higher accuracy on average.

2) At higher compression ratio (80%), LLMFolder shows substantial improvements over Wanda and RIA, achieving up to 215× and 406× lower perplexity on average, while maintaining an average of 20.1% and 19.1% absolute improvement, with up to 65.7% higher accuracy on zero-shot tasks.

3) GPTQ and AWQ generally performs the best among all experimented methods, LLMFolder can achieve an average of 90.5% of their accuracy under 80% compression ratio, and 98.9% under 50%. PowerInfer only performs well on ReLU-activation based models.

Dataset	Method	Falcon-7B			Bloomz-7B1			GPT2-XL			OPT-6.7B			Falcon2-11B	
Compression Ratio		*50%*	*70%*	*80%*	*50%*	*70%*	*80%*	*50%*	*70%*	*80%*	*50%*	*70%*	*80%*	70%	80%
	Dense		6.6			14.1			17.4			10.9			5.2
	Wanda	8.4	65.0	1489	16.6	41.2	467	19.8	175.1	1.1e5	11.2	11.2	8429	57.3	1715
	RIA	7.3	23.2	7796	15.9	34.7	481	19.4	107.8	1.7e4	10.7	72.0	2.0e4	25.4	1427
Wikitext2	PowerInfer	2.0e4	2.7e4	4.2e4	2.4e3	4.8e3	5.4e3	3.4e3	3.9e3	3.9e3	10.9	**10.9**	**10.9**	4.2e6	4.8e6
	GPTQ	**6.6**	**6.7**	**7.0**	15.2	15.3	15.7	**17.4**	**17.5**	**17.9**	**10.8**	**10.9**	13.4	**4.8**	**5.2**
	AWQ	**6.6**	**6.7**	7.1	**14.1**	**14.2**	**14.6**	**17.4**	17.6	18.2	10.9	**10.9**	11.1	**4.8**	**5.2**
	LLMFolder	6.9	7.3	13.4	14.9	16.2	387	19.3	39.9	313	11.1	11.1	11.1	5.9	11.4
	Dense		10.1			18.5			20.9			12.7			9.1
	Wanda	13.0	85.8	1592	21.4	51.4	459	23.7	193.9	9.0e4	13.9	62.9	8274.5	99.3	2448
	RIA	11.9	33.3	9636	21.0	46.9	812	23.4	108.8	1.5e4	13.7	106.8	1.8e4	38.7	2016
C4	PowerInfer	2.1e4	5.1e4	7.8e4	1.7e3	4.1e3	4.5e3	2.5e3	2.8e3	2.8e3	**12.7**	**12.7**	**12.7**	5.7e6	5.9e6
	GPTQ	10.1	10.3	11.2	20.1	20.3	21.2	**20.9**	**21.1**	22.0	**12.7**	12.9	15.9	9.1	10.4
	AWQ	**10.0**	**10.2**	**10.9**	**18.5**	**18.6**	**19.4**	21.0	**21.1**	**21.9**	**12.7**	12.8	13.1	**9.0**	**9.8**
	LLMFolder	10.7	11.7	21.8	19.1	20.3	456	24.6	27.1	484	12.9	12.9	12.9	10.9	25
	Dense		9.9			26.8			23.2			13.1			8.4
	Wanda	12.9	93.9	2292	30.1	87.1	1058	27.3	185.0	4.6e4	17.2	90.3	6212	106	2631
	RIA	11.2	36.4	1.2e4	30.3	77.6	3141	26.5	129.5	1.4e4	14.6	171.7	1.2e4	39.2	2159
PDB	PowerInfer	2.0e4	5.6e4	8.5e4	3.0e3	5.4e3	6.1e3	2.8e3	3.1e3	3.1e3	13.1	**13.1**	**13.1**	7.8e6	8.2e6
	GPTQ	**9.9**	**10.0**	**10.6**	29.0	29.7	31.3	**23.2**	23.4	24.6	**13.0**	13.3	19.4	7.9	8.5
	AWQ	**9.9**	**10.0**	**10.6**	26.8	**27.0**	**27.4**	**23.2**	**23.3**	**24.5**	13.1	13.2	13.6	**7.9**	**8.4**
	LLMFolder	10.4	11.2	23.3	**26.6**	28.1	409	30.8	34.7	268	13.4	13.4	13.4	9.5	20.1

Table 3. Perplexity of different models on three datasets, when the model's FFN block is compressed with different methods. Text in bold marks the best.

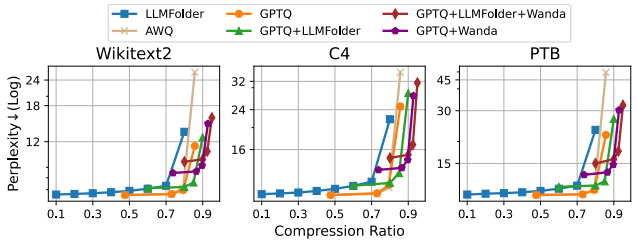


Figure 10. Perplexity (↓) comparison of AWQ, GPTQ, LLMFolder and combination of GPTQ with LLMFolder and Wanda.

Performance against Baselines. We evaluate LLMFolder against pruning (Wanda, RIA), contextual pruning (PowerInfer), and quantization (GPTQ, AWQ). Detailed results are in Table 3 (perplexity) and Table 5 (accuracy).

Unstructured Pruning (Wanda, RIA): LLMFolder consistently outperforms Wanda and RIA. At ≈50% compression, LLMFolder offers comparable or slightly better performance (e.g., Falcon-7B on Wikitext2: 6.9 vs 7.3-8.4 PPL). At higher ratios (≥70%), LLMFolder shows substantial gains; e.g., for Falcon-7B at 80% compression, PPL drops to 13.4 from 1489 (Wanda) and 7796 (RIA), while PIQA accuracy is 73.1% vs ≈54%. Wanda occasionally excels, e.g., on the Lambada benchmark at 50% compression.

Contextual Pruning (PowerInfer): PowerInfer's effectiveness is primarily on ReLU-based model (OPT6.7B). For models with other activations like Falcon, BLOOM and GPT2 (GELU), all other methods generally demonstrates better performance across various compression ratios.

GPTQ bits/ Total FFN Comress Ratio	2/ 95%	3/ 92.5%	4/ 90%	8/ 80%
PIQA	65.9%	74.3%	75.5%	76.0%
Lambda	40.3%	74.6%	76.2%	77.1%
ARC-C	22.6%	32.3%	30.9%	31.9%

Table 4. Zero-shot task accuracy of GPTQ+Wanda (50% compression)+LLMFolder (threshold=0.65) on Falcon-7B.

Quantization (GPTQ, AWQ): While GPTQ and AWQ often achieve the best results among baselines, LLMFolder remains highly competitive. On average, LLMFolder achieves 98.9% of their accuracy at 50% compression and 90.5% at 80% compression. This might positions LLMFolder as a strong alternative to pruning methods.

Overall, LLMFolder demonstrates significant advantages, particularly at high compression levels (>70%), offering a robust alternative to pruning methods. Its performance relative to leading quantization techniques is also noteworthy. LLMFolder also provides a better performance-compression trade-off than pruning, especially for non-ReLU models or when targeting very high compression ratios.

Percentage of Weights Accessed. To understand how LLMFolder's compression ratio affects the need to access original model weights during inference, we analyze the percentage of original weights accessed at different compression ratios for Falcon-7B across Wikitext-2, C4, and PTB.

Dataset	Method	Falcon-7B			Bloomz-7B1			GPT2-XL			OPT-6.7B			Falcon-2-11B	
Compression Ratio		*50%*	*70%*	*80%*	*50%*	*70%*	*80%*	*50%*	*70%*	*80%*	*50%*	*70%*	*80%*	70%	80%
PIQA	Dense		79.5%			76.4%			70.8%			76.3%			80%
	Wanda	78.1	63.3	54.1	74.5	67.7	57.6	69.4	58.8	54.6	74.4	58.2	53.7	64.5	56.2
	RIA	78.6	70.3	54.2	74.6	68.2	60	69.5	62.0	55.1	75	56.1	53.9	67.3	55.4
	PowerInfer	68.3	52.2	54.8	52.6	52.5	52.1	53.1	53.0	53.2	76.2	76.2	76.2	52.2	52.9
	GPTQ	79.4	79.0	**78.7**	76.2	75.7	74.7	**71.0**	70.8	**70.0**	**76.3**	**76.7**	75.9	79.3	78.9
	AWQ	**79.5**	**79.4**	78.6	**76.4**	**76.0**	**76.0**	70.7	**70.9**	69.9	**76.3**	**76.7**	75.6	**79.9**	**79.7**
	LLMFolder	79.1	78.4	73.1	74.5	73.2	64.3	70.1	64.2	59	**76.3**	76.3	**76.3**	79.7	75.4
Lambada	Dense		74.5%			55.9%			51.1%			67.6%			73.5%
	Wanda	78.4	35.3	0	57.8	49.8	13.4	53.8	26.1	0	70.1	20	0	13.1	0.2
	RIA	73.8	46.5	0	56.6	52.1	26.9	51.1	30.4	0	70.1	12.1	0	27	0
	PowerInfer	35.0	0	0	0	0	0	0	0	0	67.7	67.7	67.7	0	0
	GPTQ	**74.7**	74.1	72.5	55.7	55.6	53.8	50.7	**50.6**	**48.8**	67.7	67.5	66.7	73.4	71.4
	AWQ	**74.7**	74.0	**76.1**	**55.8**	**56.2**	**57.5**	**51.2**	50.1	47.7	67.7	67.5	67.6	**74.5**	**76.1**
	LLMFolder	73.0	**74.3**	61.7	51.5	49.4	12.7	42.9	20.6	0.8	**68.1**	**68.1**	**68.1**	73.2	65.7
ARC-C	Dense		40.3%			40.3%			25.1%			30.5%			50.1%
	Wanda	35.5	20.6	18.3	36.9	28.8	20.9	22.9	19.4	23	30.1	20	20.1	26.9	19.9
	RIA	37.8	26.4	20.2	36.9	29.4	21.3	23.1	18.9	21.2	30	20.8	19.9	28.2	20
	PowerInfer	30.1	22.0	19.9	18.9	19.4	19.2	20.1	20.9	20.5	30.5	30.5	30.5	21.1	22.3
	GPTQ	39.9	**39.4**	36.6	**40.4**	39.2	**39.5**	24.6	24.7	**24.9**	30.5	31.1	30.1	49.4	44.8
	AWQ	**40.1**	39.3	**38.0**	40.2	**39.8**	38.6	**24.9**	**25.6**	24,1	30.6	30.5	28.8	**49.9**	**47.6**
	LLMFolder	38.9	35.7	26.7	38.1	35.2	23.7	22.5	19.4	20.6	**31.7**	**31.7**	**31.7**	46.2	34.1

Table 5. Zero-shot accuracy (%) of tasks on different models with varying compression ratios. Text in bold marks the best.

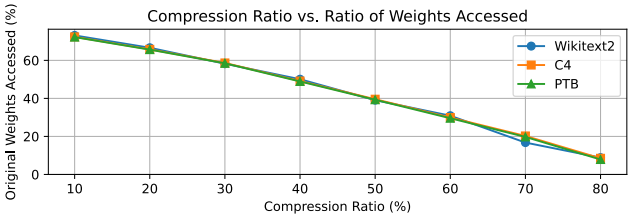


Figure 11. Percentage of original weights accessed during inference versus FFN compression ratio.

As Figure 11 shows, the percentage of original weights accessed decreases with higher compression ratios. For example, at a compression ratio of 10%, around 26% of the original weights are accessed during inference. This percentage decreases nearly linearly with higher compression ratios, reaching about 10% at 70% compression and dropping to approximately 8% at 90% compression. This trend is consistent across all three datasets, indicating that as LLMFolder compresses the model more aggressively, it relies less on the original weights, which improves inference efficiency.

7.3 Composability with Quantization and Pruning

LLMFolder is orthogonal to quantization and pruning, allowing combination for improved precision-compression tradeoffs. We evaluated LLMFolder (fixed 0.65 threshold) with GPTQ and Wanda (50% pruned), using the GPTQ-quantized W_1 FFN matrix as LLMFolder's predictor.

As shown in Figure 10 and Table 4, this combined approach offers two key benefits. First, it achieves up to 95% compression while preserving model performance, a level unattainable by individual methods. Second, it provides a superior performance-compression tradeoff. For instance, at 92.5% compression, the GPTQ + LLMFolder + Wanda configuration yields an average of 4.4% accuracy drop from the original model, whereas GPTQ alone at 86% compression (2-bit) results in a 21.8% accuracy drop. Furthermore, Figure 10 shows that combining GPTQ with LLMFolder achieves better performance tradeoffs than combining GPTQ with Wanda.

These results demonstrate that integrating LLMFolder with established methods enhances the balance between model performance and compression.

	Calibration Dataset		
Evaluation Dataset	Wikitext2	C4	PPL Change
Wikitext2	7.26	7.34	+0.08
C4	11.70	11.33	+0.37

Table 6. Perplexity (PPL) sensitivity to calibration dataset. Lower is better. The table shows the PPL when the model is evaluated on one dataset (row) after being calibrated on another (column).

7.4 Range Assignment Algorithm

The range assignment algorithm, a key component of LLMFolder (Section 5.1), determines the linear range for each neuron. This section evaluates its precision, effectiveness, and sensitivity to the calibration dataset.

Precision. Precision is assessed by comparing the actual percentage of activations within the assigned linear range

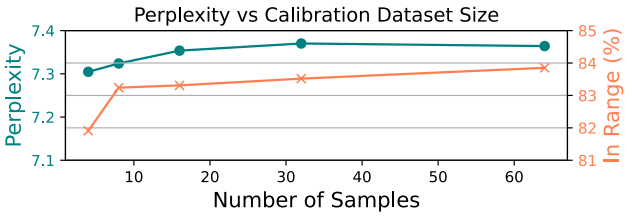


Figure 12. Perplexity (left) and actual percentage of tokens with activation input in the linear range (right) versus the number of samples in the calibration dataset.

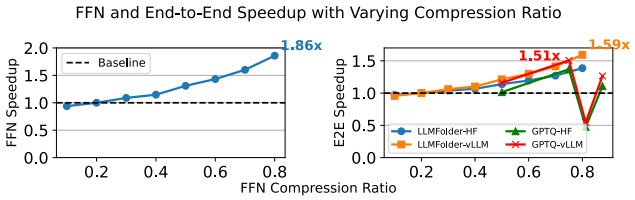
against the target threshold on the WikiText2 dataset. Figure 12 (right y-axis, orange) shows the actual percentage of activations in the linear range for a target of 85%. With only 8 samples, the actual percentage is within 1.8% of the target. Increasing sample size further refines this alignment.

Data-efficiency and Sensitivity on Calibration Set. LLMFolder is data-efficient, requiring a small calibration dataset due to its independence from backpropagation for weight updates. This efficiency was evaluated on Falcon-7B using WikiText-2. Figure 12 (left y-axis, green) shows stable model perplexity across calibration dataset sizes from 8 to 64 samples (variation < 0.06), indicating robustness to sample size. Perplexity correlates strongly with the percentage of activations within the linear range, where a higher percentage implies increased cumulative approximation error.

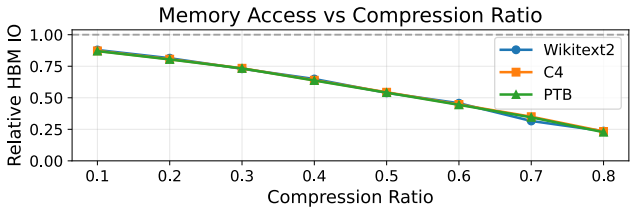
Sensitivity to calibration dataset distribution was assessed in two ways. First, we performed a cross-evaluation using WikiText-2 and C4 subsets (8 samples each), calibrating on one dataset and evaluating on the other for language generation. As shown in Section 7.3, the choice of calibration dataset source has minimal impact on performance, with perplexity variations of only 0.08 and 0.37.

Second, to test for bias from the data's statistical properties, we replaced the calibration data with randomly generated tokens from four distributions: normal, uniform, beta, and poisson. Across all distributions, the identified linear ranges were remarkably similar. For the Falcon-7B model, with ranges set to capture 65% of activation inputs, the range centers showed a standard deviation of only 0.011, and the average range length had a standard deviation of just 0.037. This low variation indicates that the location and size of the linear range are largely independent of the input distribution.

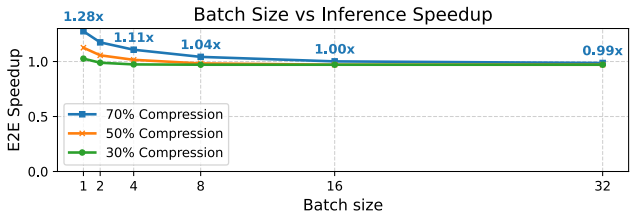
Effectiveness. The offline range assignment algorithm computes linear ranges for each neuron. For Falcon-7B (8 calibration samples, 0.85 target threshold), range search takes ≈30 minutes per layer (16 hours total). The subsequent model folding step, applying these ranges, requires only 6 seconds per layer. The offline analysis overhead for identifying hot ranges increases with model size. However, this cost is manageable because our algorithm is highly parallelizable, allowing the analysis for each neuron to be performed independently. For instance, for Falcon-40B, the analysis of hot ranges takes around 338 minutes on a dual-4090 machine



(a) Inference speedups (batch size=1)



(b) Relative Amount of HBM Memory Accesses



(c) Batch inference speedups

Figure 13. Inference Speedup and HBM Memory Access Reduction of Falcon-7B with LLMFolder

with two-way parallelism across neurons. This one-time preprocessing cost is amortized over all subsequent inferences.

7.5 Inference Speedup

We evaluate the practical inference speedup achieved by LLMFolder across different compression ratios. We first measure both FFN block-level and end-to-end speedup using the vLLM and HuggingFace frameworks on Falcon-7B. For evaluation, we perform text generation starting with 8 tokens and generating 192 additional tokens, comparing inference times across compression ratios from 10% to 80%. As comparison, we also measured GPTQ's end-to-end speedup.

As shown in Figure 13.a, LLMFolder begins providing inference benefits at 30% compression. The speedup increases with higher compression ratios, reaching 1.86× FFN speedup at 80% compression, with end-to-end inference speedups of 1.39× on HuggingFace and 1.59× on vLLM. vLLM achieves better speedup due to its optimized attention mechanism [45] - attention takes 55% of inference time in HuggingFace but only 31% in vLLM. GPTQ achieves a maximum end-to-end speedup of 1.51× at 4bit on vLLM, which is slightly lower than LLMFolder's speedup. Interestingly, GPTQ's speedup decreases with higher compression ratios (3, 2bits), probability due to the inefficieny of its implementation on modern hardwares. The speedup is mainly contributed from the reduced amount of HBM memory I/O needing during inference, as measured in Figure 13.b by manually calculating the amount of bytes that the GPU has to load from VRAM

Compression Ratio	10%	20%	30%	40%	50%	60%	70%
Inference Speedup (x)	1.49	1.72	2.02	2.46	3.13	4.32	6.97

Table 7. Inference speedup of Falcon-40B with LLMFolder at different compression ratios

to caches. The amount of I/O for the FFN blocks decreases nearly linearly as the compression ratio increases.

Batch Processing. We also evaluated LLMFolder's performance in batch inference scenarios, where multiple input sequences are processed simultaneously. We use the same setup as above, varying the batch size from 1 to 16 while keeping the total number of tokens generated constant at 200 on HuggingFace. We measured the end-to-end inference speedup at compression ratios ranging from 30% to 70%.

Figure 13c shows that as the batch size increases, the end-to-end speedup provided by LLMFolder decreases. Specifically, at a compression ratio of 70%, the speedup drops from 1.23× at batch size 1 to 0.99× at batch size 16. This trend is because the portion of original weights that need to be accessed during batch inference increases with batch size. This is because at larger batches, a neural that predicted to be in the linear range for one sequence may not be in the linear range for another sequence, necessitating access to the original weights for accurate computation. For instance, at a compression ratio of 50%, the percentage of original weights accessed increases from 35.8% at batch size 1 to 89.7% at batch size 16. This increased access to original weights reduces the performance benefits of LLMFolder in batch processing.

Similarly, when processing sequences with many initial tokens but generating few output tokens (a less common scenario [14]), LLMFolder also show limited performance gains. This is because during the initial processing phase, each input token tends to activate different neurons, reducing the sparsity patterns that LLMFolder leverages for acceleration.

Larger Models. When the model size increases to larger models that do not fit into GPU memory, LLMFolder's benefits become more pronounced. We measured the inference speedup of Falcon-40B, which requires offloading parts of the 74.5GB model parameters to CPU memory during inference. Specifically, for the baseline HuggingFace implementation, we offload part of the FFN weights to CPU memory and pin the attention weights in GPU memory. For LLMFolder, we pin all the folded matrices and predictors in GPU memory, while offloading the original FFN weights that cannot fit into GPU memory to CPU.

Table 7 shows the end-to-end inference speedup at different compression ratios. LLMFolder achieves up to 6.97× end-to-end inference speedup at a 70% compression ratio. This significant speedup is mainly due to the reduction in data transfer over the PCIe bus between CPU and GPU memory, which is substantially more expensive than accessing on-chip HBM. LLMFolder minimizes this bottleneck by ensuring only a small portion of the original weights must be transferred from the CPU to the GPU when fixing mispredicted neurons.

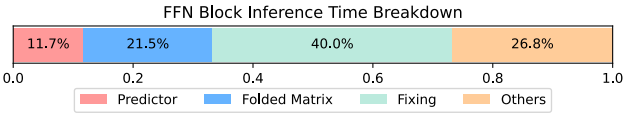


Figure 14. Performance breakdown of Falcon-7B FFN block during inference under a threshold of 0.85

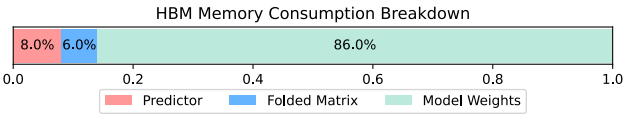


Figure 15. Memory consumption breakdown of Falcon-7B with LLMFolder.

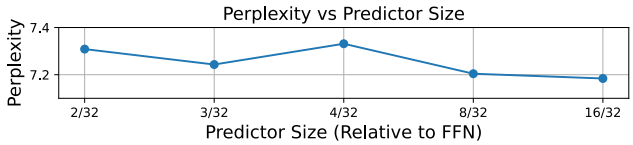


Figure 16. Influence of predictor size on perplexity in Wikitext2 dataset, evaluated with Falcon-7B.

7.6 Performance Overhead

The runtime overhead of LLMFolder arises from two main sources: additional memory usage and inference time. For Falcon-7B, each layer (32 in total) requires 39MB for the precomputed matrix and 20MB for the predictor, totaling 59MB per layer. Figure 15 breakdowns the memory consumption, showing that LLMFolder incurs a 14% memory overhead compared to the original model size (13.3GB). Inference time is primarily affected by four components: predictor execution, constant folded matrix computation, result fixing, and auxiliary operations (e.g., mask generation and index conversion for correcting mispredicted neurons). As shown in Figure 14 with a target threshold of 0.85, result fixing dominates due to costly sparse matrix indexing and loading in CUDA kernels. The predictor accounts for 11.7% of the total time, folded matrix computation consumes another 22%, and auxiliary operations the remainder. This indicates that sparse operations, rather than prediction, contribute most to the computational overhead.

7.7 Influence of Predictor Size

The predictor's accuracy in LLMFolder, influenced by its size, affects model performance [63]. We evaluated Falcon-7B's perplexity on WikiText2 with varying predictor sizes, controlled by the number of bits used in quantization [34] of the weight matrix in FFN. Figure 16 shows that perplexity decreases as predictor size increases, with a maximum difference of 0.12. This suggests that LLMFolder can use a small predictor to balance performance and memory footprint.

Intermediate Type	Original	bfloat16	float16	float32	float64
Average MSE	0	2.9e-3	4.9e-5	1.3e-5	1.3e-5
Perplexity	7.184	7.369	7.185	7.186	7.185

Table 8. Perplexity and FFN MSE on WikiText2 with and without folding by using different intermediate data types during folded matrix generation

7.8 Precision Effects on Reordering FFN

Floating-point arithmetic is commutative but not strictly associative or distributive due to finite precision and rounding errors [8]. LLMFolder leverages these properties for constant folding. We empirically evaluate the numerical impact of these approximations. Our evaluation comprises two parts: a perplexity comparison between folded and non-folded implementations, and a direct measurement of numerical differences in matrix operations at various scales.

The perplexity comparison involves three scenarios: (1) baseline sequential computation (no folding), (2) computation using the folded matrix, and (3) hybrid computation using ground-truth activation inputs for linear range determination, thereby isolating predictor effects.

Table 8 presents perplexity measurements for various floating-point precisions. Minimal perplexity differences (<0.1%) are observed between folded and non-folded implementations using float32 or float16. Notable differences arise only with bfloat16, which has lower mantissa precision.

To quantify numerical stability at varying scales, we computed the Mean Squared Error (MSE) between folded and original matrix computations for matrices of the original Falcon-7B FFN size (1×) as well as for FFNs scaled by factors of 4× and 8×. These computations consistently yielded low MSE values (all $< 10^{-6}$ across these scales), indicating that numerical errors from matrix reordering are negligible, even with increased matrix dimensions.

8 Related Works

LLM Inference Acceleration. Beyond model compression, significant research targets efficient LLM inference algorithms. Speculative decoding employs a smaller, faster LLM to draft tokens, subsequently verified by a larger LLM [22, 24, 47, 48]. FlashAttention [27, 28] optimizes attention I/O by tiling computations. Building on this, FlashDecoding [26] enhances parallelism in sequence dimensions, and FlashDecoding++ [42] further reduces synchronization overhead via an equivalent softmax transformation.

LLMFolder reduces I/O costs and improves inference speed by merging FFN matrix multiplication and activation functions into a single operation. This approach is orthogonal to existing inference acceleration techniques, allowing for potential synergistic performance gains.

Activation Function Approximation. Activation function approximation approaches are either hardware-centric or software-centric. Hardware approaches [58, 77] focus on FPGA implementations and simpler neural networks. Software approaches [50] target training optimization through piecewise linear approximation.

In contrast, LLMFolder combines activation function approximation with constant folding, enabling parameter reduction through matrix multiplication merging for both memory and speed benefits.

9 Discussion and Limitations

A key limitation of LLMFolder is its handling of LLMs employing GLU-variant [62] Feed-Forward Networks (Gated-FFN). Models such as LLaMA3.1 [31] and Qwen 2.5 [72] utilize FFN variants with quadratic operations: $FFN(x) = \sigma(xW_1) \odot (xW_2)W_3$, where $\odot$ denotes elementwise multiplication. This architecture presents challenges for matrix folding due to the exponential growth of folded matrices. For instance, applying LLMFolder to LLaMA-2-7B would increase parameters by 254×, undermining the objective of reducing computational costs.

Despite this, non-gated models remain highly popular and competitive, which demonstrates the broad applicability of LLMFolder. On HuggingFace, the most downloaded text generation model is the non-gated GPT-2 (14.3M/month). According to our analysis across 17 language model tasks on HuggingFace, non-gated models comprise nearly 75% of the top 20 downloaded models, representing 92.1% (586.6M) of their total monthly downloads. This popularity might be supported by strong performance. For instance, the non-gated Falcon-7B achieves perplexity scores (6.6 on Wikitext-2, 10.1 on C4) comparable to the gated LLaMA3-8B (6.1, 9.2).

Future research could focus on developing specialized folding techniques to efficiently manage these quadratic operations while maintaining acceptable overhead.

10 Conclusion

LLMFolder, a novel approach, compresses LLMs by applying constant folding optimization to feed-forward networks (FFNs). It approximates non-linear activations with linear functions in common input ranges, enabling parameter reduction via matrix multiplication reordering. Experiments demonstrate LLMFolder reduces FFN parameters by up to 80% (95% with quantization and pruning) while maintaining accuracy. This achieves 1.6× inference speedup on vLLM and 1.4× on HuggingFace.

Acknowledgments

We would like to thank our shepherd Yongjoo Park and the anonymous reviewers for their constructive feedback. This work is supported by the National Key R&D Program of China under grant No. 2023YFB3002000 and the National Natural Science Foundation of China under grant No. 62422209.

References

[1] [n. d.]. Constant folding - Wikipedia. https://en.wikipedia.org/wiki/Constant_folding.

[2] [n. d.]. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ [Online; accessed 2024-11-01].

[3] [n. d.]. openai-community/gpt2-xl · Hugging Face. https://huggingface.co/openai-community/gpt2-xl [Online; accessed 2025-01-09].

[4] [n. d.]. Optimize Options (Using the GNU Compiler Collection (GCC)). https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html. (Accessed on 11/29/2024).

[5] [n. d.]. python/cpython: The Python programming language. https://github.com/python/cpython?tab=readme-ov-file. (Accessed on 11/29/2024).

[6] [n. d.]. Release PyTorch 2.5.1: bug fix release · pytorch/pytorch. https://github.com/pytorch/pytorch/releases/tag/v2.5.1 [Online; accessed 2024-11-24].

[7] [n. d.]. Release v0.6.6 · vllm-project/vllm. https://github.com/vllm-project/vllm/releases/tag/v0.6.6 [Online; accessed 2025-01-14].

[8] [n. d.]. Sixth Computational and Data Science school for HEP (CoDaS-HEP 2024) (22-26 July 2024): Floating Point Arithmetic Is Not Real · Indico. https://indico.cern.ch/event/1422680/contributions/5983305/ [Online; accessed 2024-12-16].

[9] [n. d.]. Spark SQL & DataFrames | Apache Spark. https://spark.apache.org/sql/. (Accessed on 11/29/2024).

[10] [n. d.]. Transformers. https://huggingface.co/docs/transformers/index. (Accessed on 06/23/2024).

[11] [n. d.]. Welcome to cuML's documentation! — cuml 24.10.00 documentation. https://docs.rapids.ai/api/cuml/stable/. (Accessed on 12/03/2024).

[12] 2023. EleutherAI/lambada_openai · Datasets at Hugging Face. https://huggingface.co/datasets/EleutherAI/lambada_openai [Online; accessed 2025-01-09].

[13] 2023. LDS.128 loads from shared memory - CUDA / CUDA Programming and Performance - NVIDIA Developer Forums. https://forums.developer.nvidia.com/t/lds-128-loads-from-shared-memory/264465 [Online; accessed 2024-11-01].

[14] 2023. philschmid/sharegpt-raw · Datasets at Hugging Face. https://huggingface.co/datasets/philschmid/sharegpt-raw [Online; accessed 2025-01-07].

[15] 2023. tiiuae/falcon-11B · Hugging Face. https://huggingface.co/tiiuae/falcon-11B [Online; accessed 2025-01-13].

[16] 2024. allenai/ai2_arc · Datasets at Hugging Face. https://huggingface.co/datasets/allenai/ai2_arc [Online; accessed 2025-01-09].

[17] 2024. facebook/opt-6.7b · Hugging Face. https://huggingface.co/facebook/opt-6.7b [Online; accessed 2025-01-09].

[18] 2024. google/switch-c-2048 · Hugging Face. https://huggingface.co/google/switch-c-2048 [Online; accessed 2025-01-08].

[19] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The Falcon Series of Open Language Models. arXiv:2311.16867 [cs.CL] https://arxiv.org/abs/2311.16867

[20] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 7432–7439.

[21] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[22] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D Lee, Deming Chen, and Tri Dao. 2024. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774* (2024).

[23] Rahul Chand, Yashoteja Prabhu, and Pratyush Kumar. 2023. Dsformer: Effective compression of text-transformers by dense-sparse weight factorization. *arXiv preprint arXiv:2312.13211* (2023).

[24] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318* (2023).

[25] Djork-Arné Clevert. 2015. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289* (2015).

[26] Tri Dao. [n. d.]. Stanford CRFM. https://crfm.stanford.edu/2023/10/12/flashdecoding.html [Online].

[27] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691* (2023).

[28] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems* 35 (2022), 16344–16359.

[29] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. 2017. Language modeling with gated convolutional networks. In *International conference on machine learning*. PMLR, 933–941.

[30] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2024. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 2198, 15 pages.

[31] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[32] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2018. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks* 107 (2018), 3–11.

[33] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. OPTQ: Accurate quantization for generative pre-trained transformers. In *The Eleventh International Conference on Learning Representations*.

[34] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG] https://arxiv.org/abs/2210.17323

[35] O. Frantar and D. Alistarh. 2023. SparseGPT: A One-Shot Pruning Method for Neural Networks. In *Proceedings of the International Conference on Learning Representations*.

[36] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2024. A framework for few-shot language model evaluation. https://doi.org/10.5281/zenodo.12608602

[37] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[38] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2023. Knowledge distillation of large language models. *arXiv preprint arXiv:2306.08543* (2023).

[39] Jinyang Guo, Jianyu Wu, Zining Wang, Jiaheng Liu, Ge Yang, Yifu Ding, Ruihao Gong, Haotong Qin, and Xianglong Liu. 2024. Compressing large language models by joint sparsification and quantization. In *Forty-first International Conference on Machine Learning*.

[40] Mark Harris. 2014. How to Access Global Memory Efficiently in CUDA C/C++ Kernels | NVIDIA Technical Blog. https://developer.nvidia.com/blog/how-access-global-memory-efficiently-cuda-c-kernels/ [Online; accessed 2024-11-01].

[41] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[42] Ke Hong, Guohao Dai, Jiaming Xu, Qiuli Mao, Xiuhong Li, Jun Liu, Kangdi Chen, Yuhan Dong, and Yu Wang. 2023. Flashdecoding++: Faster large language model inference on gpus. *arXiv preprint arXiv:2311.01282* (2023).

[43] Young Jin Kim, Rawn Henry, Raffy Fahim, and Hany Hassan Awadalla. 2023. Finequant: Unlocking efficiency with fine-grained weight-only quantization for llms. *arXiv preprint arXiv:2308.09723* (2023).

[44] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. 2017. Self-normalizing neural networks. *Advances in neural information processing systems* 30 (2017).

[45] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[46] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2023. Bloom: A 176b-parameter open-access multilingual language model. (2023).

[47] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*. PMLR, 19274–19286.

[48] Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077* (2024).

[49] Yixiao Li, Yifan Yu, Qingru Zhang, Chen Liang, Pengcheng He, Weizhu Chen, and Tuo Zhao. 2023. Losparse: Structured compression of large language models based on low-rank and sparse approximation. In *International Conference on Machine Learning*. PMLR, 20336–20350.

[50] Xuan Liao, Tong Zhou, Longlong Zhang, Xiang Hu, and Yuanxi Peng. 2023. A method for calculating the derivative of activation functions based on piecewise linear approximation. *Electronics* 12, 2 (2023), 267.

[51] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. *Proceedings of Machine Learning and Systems* 6 (2024), 87–100.

[52] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. 2023. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*. PMLR, 22137–22176.

[53] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems* 36 (2023), 21702–21720.

[54] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn Treebank: annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology* (Plainsboro, NJ) *(HLT '94)*. Association for Computational Linguistics, USA, 114–119. https://doi.org/10.3115/1075812.1075835

[55] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843* (2016).

[56] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564* (2023).

[57] Seyed Iman Mirzadeh, Keivan Alizadeh-Vahid, Sachin Mehta, Carlo C del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. ReLU Strikes Back: Exploiting Activation Sparsity in Large Language Models. In *The Twelfth International Conference on Learning Representations*.

[58] Amit Mishra, Krishna Raj, et al. 2007. Implementation of a digital neuron with nonlinear activation function using piecewise linear approximation technique. In *2007 Internatonal Conference on Microelectronics*. IEEE, 69–72.

[59] Emanuel Parzen. 1962. On estimation of a probability density function and mode. *The annals of mathematical statistics* 33, 3 (1962), 1065–1076.

[60] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67. http://jmlr.org/papers/v21/20-074.html

[61] Rajarshi Saha, Varun Srivastava, and Mert Pilanci. 2023. Matrix compression via randomized low rank and low precision factorization. *Advances in Neural Information Processing Systems* 36 (2023).

[62] Noam Shazeer. 2020. GLU Variants Improve Transformer. arXiv:2002.05202 [cs.LG] https://arxiv.org/abs/2002.05202

[63] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. 2024. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 590–606.

[64] Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2024. A Simple and Effective Pruning Approach for Large Language Models. In *The Twelfth International Conference on Learning Representations*.

[65] George R Terrell and David W Scott. 1992. Variable kernel density estimation. *The Annals of Statistics* (1992), 1236–1265.

[66] Inar Timiryasov and Jean-Loup Tastet. 2023. Baby llama: knowledge distillation from an ensemble of teachers trained on a small dataset with no performance penalty. *arXiv preprint arXiv:2308.02019* (2023).

[67] Contributors to Wikimedia projects. 2004. Norm (mathematics) - Wikipedia. https://en.wikipedia.org/wiki/Norm_(mathematics) [Online; accessed 2024-11-13].

[68] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971 [cs.CL] https://arxiv.org/abs/2302.13971

[69] Wenxiao Wang, Wei Chen, Yicong Luo, Yongliu Long, Zhengkai Lin, Liye Zhang, Binbin Lin, Deng Cai, and Xiaofei He. 2024. Model compression and efficient inference for large language models: A survey. *arXiv preprint arXiv:2402.09748* (2024).

[70] Xiuying Wei, Yunchen Zhang, Yuhang Li, Xiangguo Zhang, Ruihao Gong, Jinyang Guo, and Xianglong Liu. 2023. Outlier suppression+: Accurate quantization of large language models by equivalent and optimal shifting and scaling. *arXiv preprint arXiv:2304.09145* (2023).

[71] Mengwei Xu, Wangsong Yin, Dongqi Cai, Rongjie Yi, Daliang Xu, Qipeng Wang, Bingyang Wu, Yihao Zhao, Chen Yang, Shihe Wang, et al. 2024. A survey of resource-efficient llm and multimodal foundation models. *arXiv preprint arXiv:2401.08092* (2024).

[72] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2. 5 Technical Report. *arXiv preprint arXiv:2412.15115* (2024).

[73] Ge Yang, Changyi He, Jinyang Guo, Jianyu Wu, Yifu Ding, Aishan Liu, Haotong Qin, Pengliang Ji, and Xianglong Liu. 2024. LLMCBench: Benchmarking Large Language Model Compression for Efficient Deployment. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[74] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in*

Neural Information Processing Systems 35 (2022), 27168–27183.

[75] Lu Yin, You Wu, Zhenyu Zhang, Cheng-Yu Hsieh, Yaqing Wang, Yiling Jia, Gen Li, Ajay Kumar Jaiswal, Mykola Pechenizkiy, Yi Liang, Michael Bendersky, Zhangyang Wang, and Shiwei Liu. 2024. Outlier Weighed Layerwise Sparsity (OWL): A Missing Secret Sauce for Pruning LLMs to High Sparsity. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 57101–57115. https://proceedings.mlr.press/v235/yin24e.html

[76] Haoran You, Yipin Guo, Yichao Fu, Wei Zhou, Huihong Shi, Xiaofan Zhang, Souvik Kundu, Amir Yazdanbakhsh, and Yingyan Celine Lin. 2024. ShiftAddLLM: Accelerating Pretrained LLMs via Post-Training Multiplication-Less Reparameterization. *arXiv preprint arXiv:2406.05981* (2024).

[77] Peter W Zaki, Ahmed M Hashem, Emad A Fahim, Mostafa A Masnour, Sarah M ElGenk, Maggie Mashaly, and Samar M Ismail. 2019. A novel sigmoid function approximation suitable for neural networks on FPGA. In *2019 15th International Computer Engineering Conference (ICENCO)*. IEEE, 95–99.

[78] Yingtao Zhang, Haoli Bai, Haokun Lin, Jialin Zhao, Lu Hou, and Carlo Vittorio Cannistraci. 2024. Plug-and-play: An efficient post-training pruning method for large language models. In *The Twelfth International Conference on Learning Representations*.

[79] Yang Zhang, Yanfei Dong, and Kenji Kawaguchi. 2024. Investigating Layer Importance in Large Language Models. In *Proceedings of the 7th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*. 469–479.

[80] Changhai Zhou, Yuhua Zhou, Shijie Han, Qian Qiao, and Hongguang Li. 2024. QPruner: Probabilistic Decision Quantization for Structured Pruning in Large Language Models. *arXiv preprint arXiv:2412.11629* (2024).

[81] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294* (2024).