

Scaling Up On-Device LLMs via Active-Weight Swapping Between DRAM and Flash

Fucheng Jia*

Central South University
Microsoft Research
fuchengjia@csu.edu.cn

Huiqiang Jiang

Microsoft Research
hjiang@microsoft.com

Yunxin Liu

Institute for AI Industry Research
(AIR), Tsinghua University
liuyunxin@air.tsinghua.edu.cn

Zewen Wu*

Tsinghua University
Microsoft Research
wuzw21@mails.tsinghua.edu.cn

Qianxi Zhang

Microsoft Research
qianxi.zhang@microsoft.com

Ju Ren

Tsinghua University
renju@tsinghua.edu.cn

Ting Cao†

Institute for AI Industry Research
(AIR), Tsinghua University
tingcao@mail.tsinghua.edu.cn

Shiqi Jiang

Microsoft Research
shijiang@microsoft.com

Yuqing Yang

Microsoft Research
yuqing.yang@microsoft.com

Deyu Zhang

Central South University
zdy876@csu.edu.cn

Abstract

Large language models (LLMs) are increasingly being deployed on mobile devices, but the limited DRAM capacity constrains the deployable model size. This paper introduces ACTIVEFLOW, the first LLM inference framework that can achieve adaptive DRAM usage for modern LLMs (not ReLU-based), enabling the scaling up of deployable model sizes. The framework is based on the novel concept of *active weight DRAM-flash swapping* and incorporates three novel techniques: (1) Cross-layer active weights preloading. It uses the activations from the current layer to predict the active weights of several subsequent layers, enabling computation and data loading to overlap, as well as facilitating large I/O transfers. (2) Sparsity-aware self-distillation. It adjusts the active weights to align with the dense-model output distribution, compensating for approximations introduced by contextual sparsity. (3) Active weight DRAM-flash swapping pipeline. It orchestrates the DRAM space allocation among the hot weight cache, preloaded active weights, and computation-involved weights based on available memory. Results show ACTIVEFLOW achieves the performance-cost Pareto frontier compared to existing efficiency optimization methods.

1 Introduction

Large language models (LLMs) are increasingly deployed on mobile and PC devices as integral system components, such as the on-device 3B Apple foundation model for Apple iOS [4], the 3.82B Phi Silica for Windows [18], and 3.35B Gemini Nano for Google’s Android [26].

However, further scaling up the on-device LLM size is very difficult, with a key constraint of DRAM size. Due to power and area constraints, the DRAM size on mobile devices remains limited and difficult to increase, even across device upgrades (e.g., both iPhone 15 and iPhone 16 feature 8 GB DRAM). Furthermore, the available DRAM capacity is also determined by the co-active apps and OS processes remaining in DRAM simultaneously. Mobile OS can terminate an app under low available DRAM unless the app can reduce the memory usage[35].

Goal. To enable the deployment of larger LLMs, it is essential to realize *adaptive DRAM usage* for LLM inference. That is, the inference process dynamically adapts to different available DRAM sizes while maintaining comparable model quality and inference speed. Mirroring the OS employs virtual memory to abstract physical limitations, this work aims for adaptive DRAM usage that is transparent to the user, creating the illusion that the entire model resides in DRAM.

Adaptive DRAM usage has been previously investigated for traditional non-autoregressive DNNs (e.g., CNN and Bert)

*Research interns at Microsoft Research.

†Corresponding author.

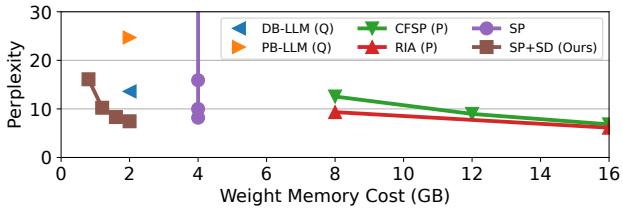


Figure 1: The perplexity versus cost of LLaMA-3-8B model. Ours shows the Pareto frontier compared with SOTA model compression methods including quantization (Q), pruning (P) and contextual sparsity (SP). Each point on the scaling line means a sparsity ratio.

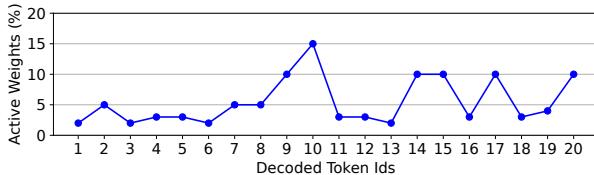


Figure 2: The upper bound sparsity of LLaMa-2-70B model during decoding.

through DRAM-Flash swapping [14, 38]. However, the fundamental difference in workload characteristics hinders the direct application of these methods to LLMs. Existing techniques rely on the computation-intensive feature of traditional DNNs, so the current operator computation can overlap the loading of the next operator. While this overlap is present in the LLM prefilling stage, the significantly more time-consuming autoregressive decoding phase is bottlenecked by memory access. Consequently, realizing user-oblivious adaptive memory management for LLM inference necessitates minimizing Flash data loading to mitigate the substantial disparity between memory and Flash bandwidth ($\sim 5\times$ on mobile phones).

Fortunately, a unique characteristic of LLMs is *contextual sparsity*, where although the model itself is large, only a small subset of weights is actively used per token generation [17], which we term as *active weights*. Our upper-bound analysis (Fig. 2) shows that during each inference iteration, only $< 15\%$ weights need to be activated to generate the same token.

Challenges. This contextual sparsity inspires us to explore the new opportunity of *active weights swapping* for adaptive memory usage. Unlike traditional per-operator swapping, active weight swapping introduces greater challenges: (1) How to accurately identify the active weights, given contextual sparsity is highly dynamic, varying cross tokens, layers and blocks. Misidentification could degrade model accuracy. (2) How to predict the active weights as early as possible,

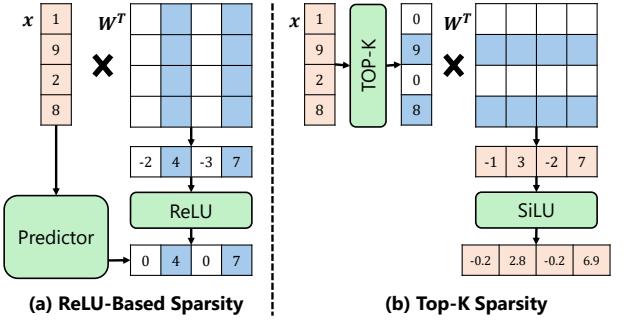


Figure 3: The ReLU-based sparsity and Top-K activation sparsity. We base our system on Top-K sparsity due to its broader applicability and higher accuracy.

allowing for overlapping computation with loading, as well as efficient large I/O transfers, both of which are critical for performance.

Several works have explored contextual sparsity [11, 17, 19, 22–24, 28, 36], but gaps remain in addressing the challenges above. Some methods like Deja Vu [17], PowerInfer [23] and LLM in a flash [2] use available ReLU-based models to generate zero activations and introduce additional predictors (GB memory cost) to forecast these zeros. However, modern LLMs used in production (e.g., LLaMA) rarely use ReLU-based architecture due to its inferior accuracy [27] (see Fig. 14b). There are also works performing continued pre-training to transform available models to ReLU or ReLU-variant based, such as PowerInfer-2 [36], TurboSparse [24], ProSparse [22], and Q-Sparse [28]. These works require training on hundreds of billions of tokens and consume substantial hardware resources. There are also works, such as InfiniGen [12], NSA [39], and SeerAttention [9], focusing on KV cache sparsity but not weight sparsity. These methods benefit long context scenarios ($> 32K$) which are not the common cases on edge. TEAL [15] proposes a training-free, magnitude-based sparsity method (see Fig. 3), where only activations above a threshold are computed. However, the active weights cannot be predicted, but only be identified after the input activation is ready. Additionally, the method is empirical, and there is no mechanism to compensate for the accuracy loss due to the potential misidentification of active weights. Therefore, current techniques fall short of achieving adaptive memory usage for LLMs.

Our work. This paper proposes ACTIVEFlow LLM inference framework. It can realize user-oblivious adaptive DRAM usage, in order to scale up the LLM sizes that can be deployed on mobile devices. Similar to TEAL, this paper utilizes magnitude-based, model-architecture-independent activation sparsity, to ensure the framework’s applicability

to modern LLMs. Beyond that, ACTIVEFLOW incorporates three novel techniques.

Firstly, **Cross-layer active weight preloading**. To address the sequential dependency issue of active weights with its input activation in order to enable computation and loading overlapping, we propose cross-layer active weight preloading. It creatively utilizes the current layer’s activation to pre-identify the next n layers’ active weights. It is based on the observation that due to the widely used residual connection, the activation magnitude distribution across layers share significant similarity ($>80\%$ shown in Fig. 4a). For the active weights that missed by pre-loading, ACTIVEFLOW loads on-demand when the actual activation is ready.

Secondly, **Sparsity-aware self-distillation**. Even the magnitude-based activation sparsity empirically has shown the superior quality compared to other sparse methods [16], it still introduces an approximation compared to the dense model. To compensate for the approximation, we propose sparse-aware self-distillation to adjust the active weights towards the dense-model output. The distillation improves both the sparsity ratio and model accuracy. The technique is inspired by and integrated with the quantization-aware self distillation [7]. Similar to this work, the self-distillation only needs several A100 GPU hours to train. The two methods can be used collaboratively for LLM deployment.

Thirdly, **DRAM-flash active weight swapping pipeline**. The pipeline reorganizes the data layout for the cross-layer preloading, and overlaps the active weight loading with the current layer computing. It also integrates a contextual hot active weight caching policy beyond naive swapping. The pipeline orchestrates the space allocations among the cache, preloaded active weights, and computation involved weights according to available memory.

We implement ACTIVEFLOW and evaluate it on different mobile phones (OnePlus 12, Pixel 6, and Infinix Zero). Results (Fig. 1, more in Sec. 7) show that ACTIVEFLOW achieves the **inference performance-cost Pareto frontier** among existing efficiency optimization methods, including state-of-the-art quantization (DB-LLM [6] and PB-LLM [20]), pruning (CPSP [32] and RIA [37]), and contextual sparsity (TEAL [15]), demonstrating its practical value. Particularly, under the same model quality and speed, ACTIVEFLOW reduces the DRAM usage by up to 40% for LLaMA 7B compared to llama.cpp. Under the same sparsity ratio, ACTIVEFLOW can reduce memory by 2x compared to TEAL. ACTIVEFLOW is the first to successfully deploy the original Mixtural-8x7B 4bit model [10] (no ReLU introduced) on a mid-range pixel-6 phone, achieving 1.8 tokens/s with 2.9 GB memory cost.

To summarize, the contributions of this paper are:

- We propose ACTIVEFLOW, the first LLM inference system to enable user-oblivious adaptive DRAM usage

through active weight swapping for modern general LLMs without ReLU dependency.

- We propose the cross-layer active weights preloading to allow computation/loading overlapping and large I/O transfer.
- We propose sparsity-aware self distillation to compensate the approximation introduced by sparsity.
- We implement the end-to-end ACTIVEFLOW. Results show it achieves the inference quality-cost Pareto frontier among existing optimization methods.

2 Motivation and Background

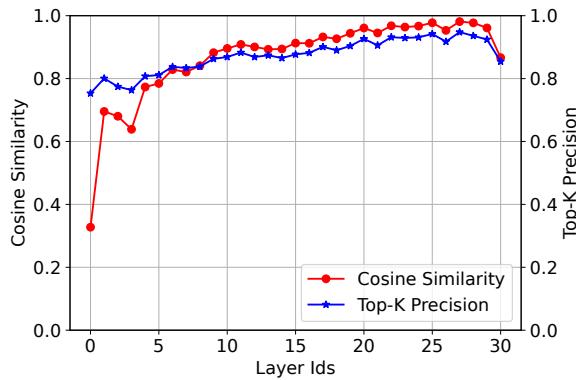
2.1 Upper Bound Analysis of Contextual Sparsity in LLMs

A specific feature of LLMs is *contextual sparsity* [11, 17, 19, 23, 24, 28, 36], which means a small, context-dependent subset of total weights, that can generate the same output as the full model. We term this small subset of weights as *active weight*. Compared to the static sparsity from model pruning [8, 25], contextual sparsity dynamically selects different active weights for computation during each token generation, preserving the model’s overall capacity and adaptability. Contextual sparsity has also been empirically demonstrated to be compatible with model quantization [28].

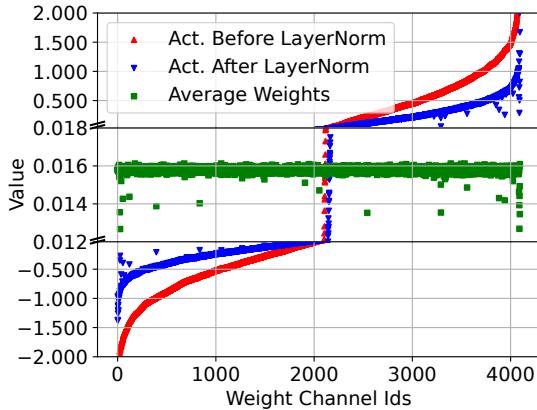
Since our techniques will be based on contextual sparsity, we first analyze the upper bound of this sparsity. We use a Llama-2-70B model to evaluate the amount of active weights required to generate the same token with full weights during the decoding process. The evaluation is conducted by incrementally removing unimportant weights for each decoded token by 1%. The important scores of weights are calculated by $S_{ij} = |W_{ij}| \cdot |X_j|$, where W_{ij} is an element of weight matrix and X_j is an element of the input activation vector. As shown in Fig. 2, the results indicate that most tokens require less than 5% of the weights, with the maximum active weight being only 15%. This high level of sparsity shows a great potential for reduced inference cost.

Although the above results are promising, it is challenging to identify the active weights during inference, unless the weights are loaded and computed with activations. Consequently, some works [17, 22] rely on ReLU-generated sparsity and propose extra predictors to estimate the sparsity, as illustrated in Fig. 3(a). These predictors are trained with calibration datasets, loaded into memory, and executed before performing per-layer LLM computations. However, the deployment cost of predictors is significant because (1) the datasets may not be suitable for real user data, (2) predictors require additional memory (at the GB level), and (3) they introduce extra computational overhead.

More recent works [16, 28] propose magnitude-based activation sparsity, as shown in Fig. 3(b). We term this sparsity



(a)



(b)

Figure 4: The cross-layer input activation similarity of a LLaMA-2-7B model. (a) The attention input cosine similarity and Top-K precision. (b) The value of activation before/after LayerNorm layer and average weights.

as *Top-K sparsity* following [28]. Only the activation elements with a magnitude above a threshold will be computed for each operator. Top-K sparsity demonstrates obvious advantages: 1) compatibility to modern non-ReLU LLMs; 2) applicability to all linear transformation operators rather than just FFN blocks; 3) no extra predictors needed.

These advantages motivate us to identify active weights for swapping based on Top-K activation sparsity.

2.2 Observation: Similarities in Cross-Layer Activations

A key observation of this paper is that **the input activations of the attention and MLP blocks in LLMs exhibit high**

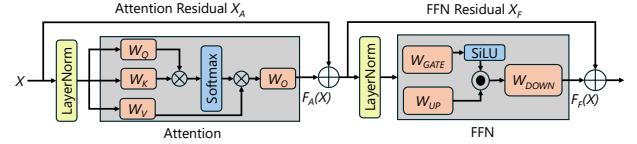


Figure 5: The simplified transformer layer structure of an LLM model. Residual connections pass the input of a block directly to output.

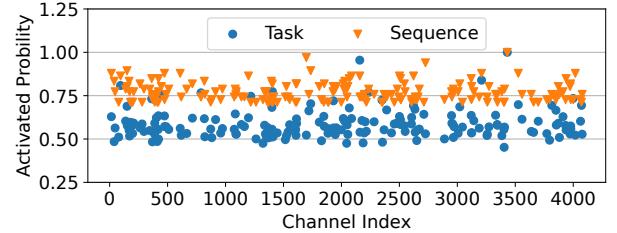


Figure 6: The selection probability of active weights in attention Q/K/V operators of Llama-2-7B model (under 50% contextual sparsity). Context level shows higher selection probability than task level. We only show the active weight with probability > 0.7.

cross-layer similarity. Fig. 4a uses the input activation of the attention block as an example to show the cosine similarity and Top-K sparsity precision in each consecutive two layers in a Llama-2-7B model. Starting from the 3rd layer, the attention Q, K, V, and FFN gate and up operators exhibit over 95% similarity. Consequently, the Top-K sparsity precision for these operators exceeds 80% cross layers.

The similarity is primarily due to the significant contribution of the residuals to the input activations. Fig. 5 shows a simplified transformer layer structure. The input activations are composed of the sum of two elements: the output activation of the previous block $F(X)$ and the residual X . The cross-layer similarity is because the residual values X are larger than the output activation values $F(X)$. This difference in values arises from (1) the LayerNorm layer in the attention and MLP blocks, and (2) the weights magnitudes. As shown in Fig. 4b, the LayerNorm reduces the activation magnitude by 50%. Additionally, the weight magnitude is smaller than the activation magnitude, resulting in a smaller calculation output.

The cross-layer input similarity motivates us for cross-layer preloading, which uses current layer's activation to identify following layers' active weights.

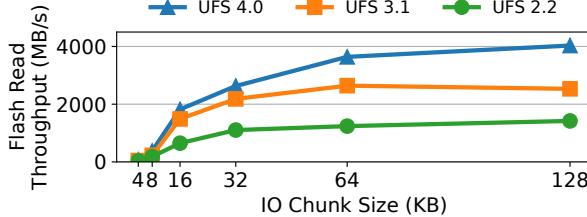


Figure 7: The flash read throughput of various IO chunk sizes on three devices with difference UFS capabilities.

2.3 Observation: Contextual Hot Active Weights During Decoding

This section investigates the presence of *hot active weights*, i.e., the weights that are frequently selected across inference iterations during decoding. This investigation aims to identify opportunities for caching and more intelligent swapping strategies. Our observation is that **contextual active weights exhibit high temporal locality across inference iterations during decoding**, suggesting that caching hot active weights for higher cache hit rates.

As shown in Fig. 6, we conducted two levels of active weight selection frequency analysis: *task level* and *context level*. The task level counts the frequency with which weight channels are selected during the decoding process for all input contexts across a dataset (WikiText-2). In contrast, the context level counts the frequency of weight selection specifically for the decoding process of a given input context. Results show that hot weight selection probabilities on the context level exceeds 0.7, while the task level exceeds 0.5. The difference demonstrates the potentially improved cache hit and reduced loading cost by implementing a contextual cache management policy.

3 Cross-layer Active Weight Preloading

To realize adaptive DRAM usage, two critical challenges for performance is: (1) whether the weight loading and computation can be overlapped to hide the flash loading overhead; (2) whether the I/O transfer can fully utilize the flash bandwidth. As shown in Fig. 7, the flash read throughput varies greatly with the chunk size of each I/O transfer. To achieve the peak flash throughput, the chunk size has to >64 KB. However, active weight from Top-K activation sparsity is in channel granularity, e.g., 4KB (see Fig. 3), and naive loading of the each active weight channel from flash can reduce the throughput from GB/s to MB/s.

However, current works including PowerInfer [23, 36], LLM in Flash [2] and Ripple [31] only partially alleviated the problem. To enlarge the chunk size, they cluster co-active

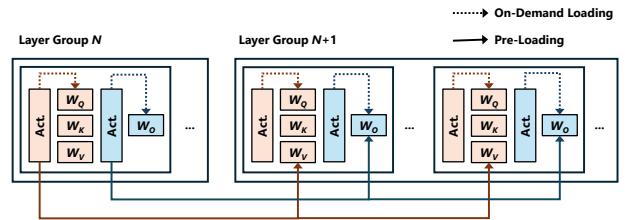


Figure 8: Cross-layer active weight pre-loading. While the computing of current layer, the active weights of all the operators in the next N layers (layer group) will be preloaded based on the current activation. The missed active weights during preloading will be on-demand loaded after its actual activation is ready.

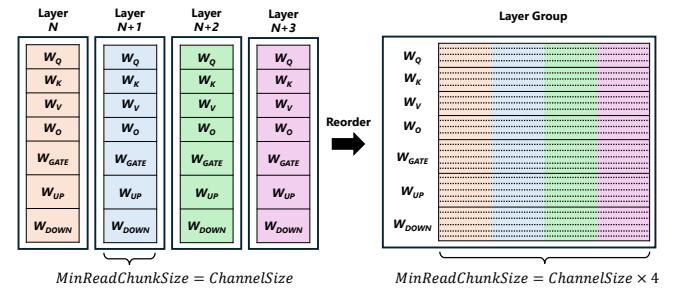


Figure 9: The reordered weights in a 4-layers group. The weight layout now is in the order of weight channel, layer, and operator type. By multi-layer weight reordering, the minimal loading chunk size is increased to improve the loading efficiency.

weight channels within the same block, and overlap each cluster loading and computation.

Our technique. To overcome the challenges, based on our key observation that cross-layer activations exhibit significant similarity, we propose the cross-layer active weight preloading. As shown in Fig. 8, while the computing of current layer, the next N layers' active weights will be preloaded to DRAM simultaneously. We term these N layers as a *layer group* for preloading. The N is set based on the available DRAM, and the computing latency ($N=4$ can fully overlap the loading and computing in our evaluation). The preloading will include the active weights from all the operators in both Attention and FFN blocks. Different activations correspond to different parts of the weights being loaded. For example, Q, K, and V activations are only used to load W_q , W_k , and W_v , respectively.

Since cross-layer activation similarity is not 100%, pre-loading can only load a portion of the necessary weights in advance. Any remaining weights that were not correctly pre-loaded are fetched through on-demand loading. This only takes $\sim 5\%$ of the total active weights.

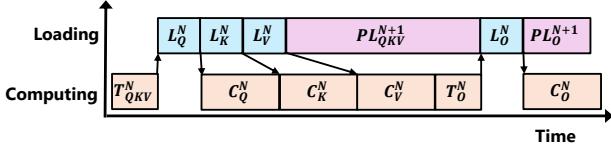


Figure 10: The computing-loading overlap pipeline for LLM inference in an attention block after warming up.

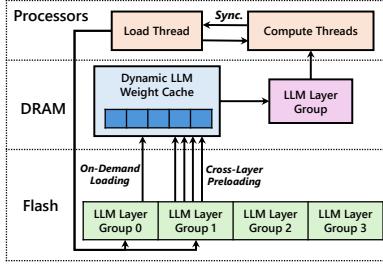


Figure 11: The weight layout and flow of ACTIVEFLOW.

Data layout. To facilitate the cross-layer preloading, the weight layout in flash is reordered, to break the tensor and layer boundary. As shown in Fig. 9 (left), the normal LLM weight layout is to arrange each weight tensor sequentially for all the operators within each layer. It is inefficient for channel-wise active weight loading. Our approach reorders the weight channels within a preloading layer group according to the order of the channel ID, layer ID, and operator type. For example, W_q weight layout in the layer group is $[Ch\ 0_{layerN}, Ch\ 0_{layerN+1}, Ch\ 0_{layerN+2}, Ch\ 0_{layerN+3}, Ch\ 1_{layerN}, Ch\ 1_{layerN+1}, Ch\ 1_{layerN+2}, Ch\ 1_{layerN+3}, \dots]$. This reordering enables pre-loading multiple layers’ weights for the same channel in a single read operation, significantly increasing the loading chunk size and improving loading efficiency.

4 Active Weight Swapping Pipeline

Building on the proposed cross-layer-group LLM weight loading and reordering techniques, we design a LLM computing-loading overlapping execution pipeline as shown in Fig. 10. The pipeline consists of four main operations: (1) Computing (C) – Performs the required computations. (2) Top-K (T) – Extracts the Top-K mask from activations to determine the indices of the activated weight channels. (3) On-demand loading (L) – Loads weights for the current layer group. (4) Preloading (PL) – Preloads weights for the next layer group.

Fig. 11 demonstrates the weight layout and flow with the pipeline. The whole model resides in the flash with the cross-layer group layout. The current active weights, as well as the pre-loaded and cached weights store in the DRAM. The computation and loading are concurrently executed.

Table 1: The symbols of our system cost model.

Symbols	Description
sp	sparsity of LLM
hr	average hit rate of weight cache
si	average similarity of cross-layer group
BW_{mem}	bandwidth of memory
BW_{small}^{flash}	bandwidth of small chunk reading from flash
BW_{large}^{flash}	bandwidth of large chunk reading from flash
S_m	Size of LLM
S_l	Size of a LLM layer
N	Layer number of a cross-layer group
M	Memory cost of pipeline
M_{max}	Memory budget
M_{cl}	Memory of a cross-layer group
M_{cache}	Memory of weight cache
M_{kv}	Memory of KV cache
T_{decode}	Decoding time of a token
T_{load}	Loading time of a cross-layer group
T_{comp}	Computing time of a cross-layer group
$T_{overlap}$	Overlapping time of two cross-layer groups
T_{onload}	On-demand loading time of a cross-layer groups
$T_{preload}$	Preloading time of a cross-layer groups

The overlapped LLM execution pipeline follows two key principles: (1) Maximize the overlap between loading and computing to minimize idle time (bubbles) to fully utilize the memory bandwidth and computing power simultaneously. (2) Maximize the cache hit rate on the sequence level. The challenge is how to accurately estimate the impact of system parameters, such as sparsity, memory cost and cache size on the accuracy and latency of model inference.

4.1 Elastic and Optimized LLM Execution

The goal of this technique is to determine the optimal system parameters, including LLM sparsity, layer number of a cross-layer group, and cache size, for a given mobile device (i.e., with specific computational power and memory budget) and a given LLM. The objective is to minimize system latency while respecting the memory constraint.

There is tradeoff between LLM sparsity, layer number of a cross-layer group and cache size on the inference metrics in terms of both latency and accuracy. Optimizing one metric could worsen another. To capture this, we define the following problem, with the memory cost as a hard constraint and the objective to minimize the decode latency. The related symbols are listed in Table 1.

$$\text{Minimize } T_{decode} = T_{load} + T_{overlap} + T_{comp} \quad (1)$$

$$M \leq M_{max} \quad (2)$$

The decode latency consists of three components: the first cross-layer-group loading time T_{load} , the cross-layer-group overlapping time $T_{overlap}$, and the final cross-layer-group computing time T_{load} , as in Eq. 1. The loading time T_{load} is the weights missed in the cache divided by the flash loading bandwidth as BW_{flash}^{small} , as in Eq. 3. The final cross-layer-group computing time T_{comp} is the group memory size M_{cl} divided by the memory bandwidth BW_{mem} , as in Eq. 4. Furthermore, the overlapping time consists of two parts, i.e., the on-demand loading time T_{load} and preloading latency $\max(T_{preload}, T_{comp})$, as in Eq. 5. We load the weights that are dissimilar across layers but not present in the cache, with latency T_{load} , as in Eq. 6. These weights typically have small chunk sizes, leading to lower bandwidth BW_{flash}^{small} . Preloading, on the other hand, loads weights at the cross-layer-group level, fetching only the cache-miss weights (Eq. 7). Since the chunk size in this stage is relatively large, the reading efficiency is significantly higher with bandwidth BW_{flash}^{large} .

$$T_{load} = \frac{M_{cl} \cdot (1 - hr)}{BW_{flash}^{small}} \quad (3)$$

$$T_{comp} = \frac{M_{cl}}{BW_{mem}} \quad (4)$$

$$T_{overlap} = T_{onload} + \max(T_{preload}, T_{comp}) \quad (5)$$

$$T_{onload} = \frac{S_l \cdot (1 - sp) \cdot (1 - hr) \cdot (1 - si)}{BW_{flash}^{small}} \quad (6)$$

$$T_{preload} = \frac{M_{cl} \cdot (1 - hr)}{BW_{flash}^{large}} \quad (7)$$

The memory cost also consists of three components: cross-layer group memory M_{cl} , weight cache memory M_{cache} , and KV cache memory M_{kv} (Eq. 8). For the KV cache, we only consider the fixed-size case. Therefore, only the first two components will dynamically influence the memory cost. The cross-layer group memory is the size of active weights, as in Eq. 9.

$$M = M_{cl} + M_{cache} + M_{kv} \quad (8)$$

$$M_{cl} = S_l \cdot (1 - sp) \cdot N \quad (9)$$

Preload-and-computation-balanced cross-layer group search. We determine the parameters (sp , S_{cl} , and M_{cache}) in a greedy manner, as follows. First, since LLM accuracy is only related to LLM sparsity, we set LLM sparsity by $sp = 1 - (M_{max}/S_m)$ to ensure the highest accuracy. Second, minimize the decode time recursively. We increase layer number of cross-layer group L in a step by step manner. This brings lower $T_{preload}$. In case $T_{preload} \leq T_{comp}$, then stop. Furthermore, if the $T_{preload}$ decrement is less than a threshold, then stop.

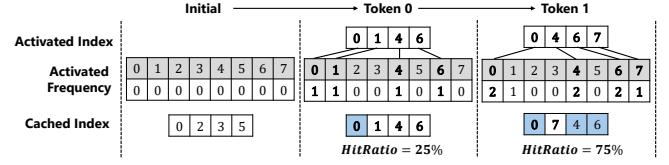


Figure 12: An example of dynamic weights caching during LLM decoding. There are 8 channels in a weight but only half of channels are cached in memory.

This approach ensures near-full memory utilization, minimal latency, and high accuracy. In case that the memory budget changes in online phase, we tune cache size to maintain well overlap between computation and flash read operations.

4.2 Dynamic LLM Weight Caching

To further reduce the number of loaded weights, we design the dynamic LLM weight caching based on observations of hot weights, as illustrated in Fig. 12. To maximize the cache hit rate, we track the frequency statistics of activation and evict the least-used weights in online phase.

To manage weight eviction, we maintain independent counters for the weights of each layer, ensuring a balanced cache size across all weights. If a newly activated channel has a higher count than the least-used channel in the cache, we evict the least-used channel. Fig. 12 illustrates an example of our dynamic cache mechanism. For a given sequence, we begin by initializing the usage count of all channels to zero. For the first token, channel index 0 is present in the cache, while channel indices 1, 4, and 6 need to be loaded from flash storage, resulting in a hit ratio of 25%. For the second token, channel indices 0, 4, and 6 hit in the cache, while only channel index 7 needs to be fetched from flash. Since channel index 1 has the lowest frequency, we replace it with channel index 7, improving the hit ratio to 75%.

5 Self-Distillation for Top-K Sparse LLM

Even with superior quality compared to other sparsity techniques, Top-K activation sparsity still introduces an approximation in active weight selection, especially in high sparsity. Traditional methods such as supervised fine-tuning often fail to recover the performance of the model under high sparsity, as they cannot effectively capture nuanced weight distributions and activation patterns caused by sparsity, leading to a degradation of precision. To address this, we propose **Top-K sparsity-aware self-distillation**, an extension of quantization and fine-tuning pipelines. It preserves the efficiency benefits of sparsity while substantially reducing computational overhead and enhancing both accuracy and generalization. In practice, it improves performance with

only a few to tens of GPU hours on a few thousand samples, and generalizes effectively across different sparsity levels.

Self distillation. As shown in Fig. 13, we maintain the outputs of the dense (teacher) and sparse (student) models, using the soft output distribution of the dense model as supervision. This allows the sparse student to capture richer correlations than hard labels and preserve fine-grained distributional details, which is crucial to compensate for information loss induced by Top-K activation sparsity.

KL loss. We adopt the **Kullback-Leibler Divergence (KLD) loss** to measure the discrepancy between the student and teacher distributions:

$$\mathcal{D}_{\text{KL}}(P_T \parallel P_S) = \sum_i P_T(i) \log \frac{P_T(i)}{P_S(i)} \quad (10)$$

Minimizing \mathcal{D}_{KL} encourages the sparse model to closely mimic the dense teacher’s distribution, preserving essential weight correlations and improving performance under high sparsity. Our framework is also orthogonal to quantization, as the KL-based distillation loss depends only on the output distribution and thus remains fully compatible with QAT, making it complementary to quantization errors and enabling additional efficiency gains with minimal accuracy loss.

Gradient STE. Gradient vanishing is a common issue when fine-tuning activation-sparse models, as the sparsity mask sets many elements to zero, preventing gradients from being properly propagated and slowing or even blocking convergence. To mitigate this, we employ **gradient Straight-Through Estimation (STE)**, which replaces the gradient of the masking operation with an identity function during backpropagation:

$$\text{forward: } y = \text{Mask}(x) \quad (11)$$

$$\text{backward: } \frac{\partial y}{\partial x} = I \quad (12)$$

This allows gradients to flow as if the mask contains non-zero values, ensuring sufficient update signals even under high sparsity. Consequently, STE accelerates convergence, enhances training robustness, and helps the model preserve critical weight distributions and activation patterns.

Inherent adaptability. A key advantage of our self-distillation framework is its inherent *adaptability across sparsity levels*.

In conventional fine-tuning or distillation pipelines, models at different sparsity ratios typically require separate training processes, which is both time-consuming and computationally expensive.

In contrast, our method requires only a *single distillation* at a fix sparsity level, where the student is forced to maintain performance under extreme information constraints. The distilled model not only captures the dense teacher’s distributional characteristics but also develops robustness

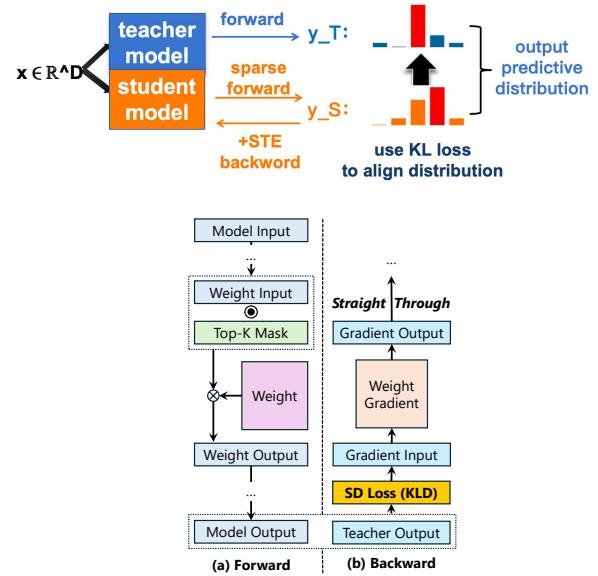


Figure 13: The forward and backward of self-distillation between teacher and student.

in reconstructing critical features, enabling direct inference across other sparsity levels without additional fine-tuning. This robustness naturally extends with increasing activation budget: model leverages the richer signals without retraining, achieving near-lossless performance over a wide sparsity range. We refer to this property as **one-distill-all-scale**; as Table 3 shows, PPL error remains within 1% even when training and inference sparsity differ by up to $\pm 15\%$.

By eliminating the need for repeated training at each sparsity level, this approach greatly reduces overall cost while ensuring consistent accuracy and efficiency in sparse LLM deployment.

6 Implementation

ACTIVEFlow is built on *llama.cpp*, a widely-used LLM inference framework for mobile devices. The whole model is stored in flash and only active weight, cached weight, and preloaded weight are in DRAM. This paper is based on the CPU backend of *llama.cpp*. The big cores execute computations and the little cores execute data loading concurrently. Since decoding speed is memory bandwidth bound, and mobile devices use a unified DRAM among all processors, we believe implementing ACTIVEFlow on different processors should have similar results. Past work [30, 34] have also demonstrated the superior performance of CPU over NPU for decoding on devices. We thus choose CPU in this paper for implementing convenience.

Flash loading. To implement cross-layer-group LLM weight loading, we modify the way weight tensors are stored in the GGUF format. Specifically, we save each operator’s weights as fundamental tensors organized in a cross-layer-group manner. We utilize *IO uring*, a low-overhead asynchronous I/O mechanism, to read the weights efficiently. In particular, we use the *io_uring_prep_read* and *io_uring_submit* functions to asynchronously request reads for active weights. After submitting all read requests, we synchronize the I/O operations using the *io_uring_wait_cqe* function. When reading active weights, we sparsely load different channels into a dense buffer, which helps optimize memory buffer layout for better compactness. Additionally, to ensure compatibility with quantization, we apply a transpose operation to the weights. This allows for complete retrieval of the necessary scaling factors when reading channels, thereby facilitating the quantization.

Swapping pipeline. To implement the active weight swapping pipeline, we first create a dedicated weight loading thread using the *ggml_thread_create* function. This thread is bound to a little core of the CPU via the *sched_setaffinity* function to optimize resource utilization. Synchronization between the weight loading thread and the main computing thread is achieved through atomic semaphores. We use *atomic_load_explicit* and *atomic_store_explicit* to manage a request signal and a complete signal that facilitates communication between the two threads. The signals operate at the cross-layer-group granularity, ensuring proper execution order between computing and weight loading operations.

Caching. Additionally, we implement the dynamic LLM weight caching, where caching is managed separately for each weight tensor. We use a hash table-based approach to efficiently query cached weight channels and dynamically track their activation frequency during decoding. When loading a new channel, we replace the least frequently activated channel, updating its index pointer in the hash table accordingly. Furthermore, we develop a kernel for generating active channel indices. This kernel maintains activation thresholds corresponding to different LLM sparsity levels. Before each activation step, it determines whether a channel should be activated based on the appropriate threshold.

Self-distillation. In order to implement the sparsity-aware self-distillation, we develop a plug-and-play sparse module in BitDistiller [7], an open-source framework for quantization-aware LLM distillation. Specially, we insert an activation sparsity module before each LLM weight computation. This module preloads a sparsity threshold for the activations and generates a Top-K mask at inference time by comparing the activations against the threshold. During backpropagation, we incorporate a gradient STE layer for each LLM weight. In addition, we implement the KLD loss function. In our self-distillation experiments, we use a sub-dataset from C4

dataset, with each epoch containing approximately 50K data samples (10B tokens). Full self-distillation comprises two epochs, with a learning rate of 1×10^{-6} and 4-bit quantization. On 4×80G-A100, it takes approximately 10 hours for an LLM.

Overall, ACTIVEFLOW comprises 3762 new lines of C++ code and thousands lines of Python code.

7 Evaluation

We evaluate ACTIVEFLOW on both end-to-end and technique performance, compared to several baselines. The evaluation setup is as follows:

7.1 Evaluation setup

Hardware devices. As shown in Table 2, we evaluate ACTIVEFLOW on three mobile devices, covering a range from high-end to low-end. For clarity, we label the three devices as Device 1, Device 2, and Device 3.

Models. To assess end-to-end performance, we test popular LLMs, including the Llama and Mixtral series, with model sizes ranging from 7B to 56B parameters. All LLMs undergo 4-bit quantization using *Q4_0*, a widely used technique that has minimal impact on accuracy. For the technique evaluation, we extract and use eight layers from the original LLM.

Baselines. We compare ACTIVEFLOW against *llama.cpp* in terms of decoding speed and memory usage. For perplexity and accuracy evaluation, we use the original LLM, ProSparse, and TEAL as baselines. ProSparse and TEAL represent state-of-the-art ReLU-sparse and Top-K-sparse LLMs, respectively.

Measurement. Our evaluation focuses on decoding speed, perplexity, accuracy, latency, hit rate, memory cost, power, and energy consumption. We use the *clock_gettime* function to record start and end timestamps, computing latency as the difference between them. We measure the total number of decoded tokens and the total decoding time, calculating speed as $N_{tokens}/Latency$. We use *lm-eval-harness*, a widely used LLM evaluation framework, to measure perplexity on the WikiText-2 dataset and accuracy on five downstream tasks: 5-shot MMLU, 5-shot GSM8K, 25-shot ARC Challenge, 25-shot ARC Easy, and 0-shot PIQA. We track cache hits and misses, computing the hit rate as $N_{hit}/(N_{hit} + N_{miss})$. We analyze memory cost using the Android Studio Profiler. We obtain current and voltage values by reading system files (*voltage_now* and *current_now*) to calculate power consumption. These values are collected every 0.5 seconds on average, and we use the decoding latency to compute the overall energy consumption.

Table 2: The hardware devices for evaluation.

Device	CPU	Memory	Flash (MaxBW)
OnePlus 12	X4+A720+A520	16GB	UFS 4.0 (5.8 GB/s)
Pixel 6	X1+A76+A55	8GB	UFS 3.1 (4.2 GB/s)
Infinix ZERO 30	A76+A55	8GB	UFS 2.2 (3.6 GB/s)

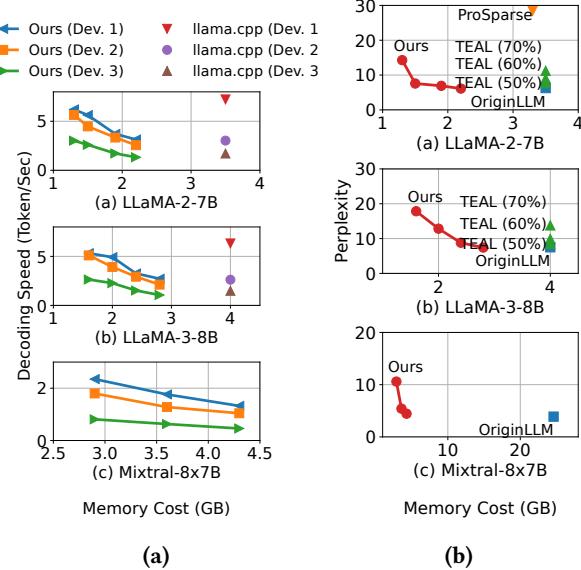


Figure 14: The end-to-end decoding speed, perplexity and memory cost of three LLMs compared with baselines on various devices. Each point represents a sparsity ratio: from left to right 0.8, 0.7, 0.6, 0.5. Since decoding is memory bound, latency increases with less sparsity and more memory cost.

7.2 End-to-end performance

Decoding speed. We first evaluate the decoding speed of different LLMs across various devices under different memory cost conditions as illustrated in Fig. 14a. For Device 2 and Device 3, using the LLaMA-2-7B model, we achieved the same performance as the full-weight memory setting while reducing memory cost by 40%. When reducing memory cost by 75%, our method achieved a 1.9× and 1.5× speedup compared to the full-weight in-memory setting on Device 2 and Device 3, respectively. The speedup is primarily due to our computing-loading pipeline, which enables higher decoding speed even under lower memory cost constraints. However, on Device 1, when using 60% of the memory cost, our performance dropped by 54% compared to the full-weight memory setting. This is because the CPU compute bandwidth of Device 1 is significantly higher than its flash read bandwidth, making the pipeline constrained by flash bandwidth. Nonetheless, at 75% memory cost, our method was able to achieve a decoding speed of 5.9 tokens per second.

For the Mixtral model, we successfully enable decoding under 6GB of memory. When the memory cost was 4.3GB, the decoding speed on Device 1, Device 2, and Device 3 was 1.3, 1.0, and 0.4 tokens per second, respectively. As the memory cost was reduced to 2.9GB, the performance improved to 2.3, 1.8, and 0.8 tokens per second, achieving a 1.8× to 2.0× speedup across the three devices.

Perplexity and Downstream tasks. Our method demonstrates that large language models can maintain low perplexity under significantly reduced memory costs, e.g., achieving performance comparable to the full-weight setting for LLaMA-2-7B and LLaMA-3-8B at only 60% memory usage in Fig. 14b, and matching the Mixtral-8x7B baseline (24.6GB) with just 4.4GB. While perplexity increases under more aggressive sparsity, our self-distillation strategy effectively alleviates performance degradation, enabling consistent improvements over TEAL across five downstream tasks, with gains up to 10.98% at 70% sparsity and average improvements ranging from 2.64% to 10.21% across sparsity levels in Table 4.

ablation study for self-distillation. To validate the effectiveness of each component in our framework, we conducted ablation studies focusing on the gradient straight-through estimator (STE) and self-distillation techniques. We carried out experiments on the Llama-3-8B model, comparing performance under different configurations, including: 1. removing STE; 2. replacing self-distillation with full fine-tuning. As shown in Table 5, our framework’s components improve model performance across different sparsity levels.

different models for self-distillation. We evaluated the proposed self-distillation framework across diverse model architectures and visualized the results in equivalent memory–performance plots. The method demonstrates strong generality: from standard 7B models to highly compressed Qwen2.5-0.5B and sparse MoE architectures, it consistently delivers significant sparsity-driven performance gains. Unlike conventional compression, our approach ensures predictable accuracy loss while achieving strict acceleration. As illustrated in Figure 15, our results consistently lie on the Pareto frontier of equivalent memory and performance, underscoring the framework’s adaptability for practical deployment.

7.3 Technique breakdown

To validate the effectiveness of our system’s techniques, we conduct ablation studies and standalone tests for each component, evaluating their impact on decoding speed, perplexity, and hit rate.

Cross-layer-group pipeline. First, we examine the effect of the cross-layer-group pipeline on decoding speed, as shown in Fig. 16. We used a 60% sparsity LLaMA-2-7B model and tested it across three devices. Our baseline consisted of

Table 3: End-to-End PPL results under Varying Sparsity Levels of Meta-LLaMA-3-8B (distillation under Fixed-Sparsity), using 4-bit quantization.

Method	0%	50%	60%	70%	80%
TOP-K	6.6836	8.1950	10.0121	15.9046	96.3015
Ours	—	7.4510	8.3216	10.2442	16.1081
Ours distill on 50%	—	7.4510	8.5625	11.8981	41.5303
Ours distill on 60%	—	7.4636	8.3216	10.8789	27.2720
Ours distill on 70%	—	7.8163	8.4440	10.2442	19.6981
Ours distill on 80%	—	9.3462	9.8854	11.1767	16.1081

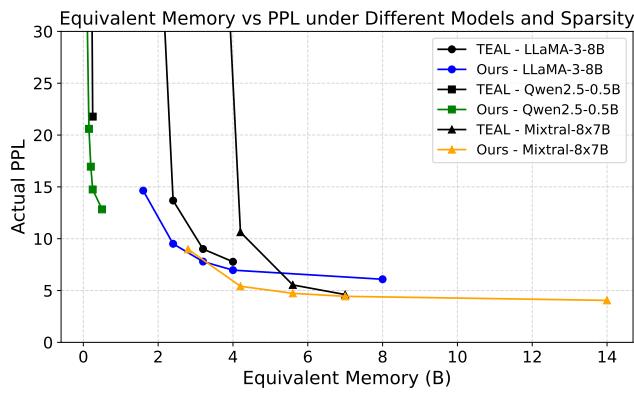


Figure 15: Pareto frontier of actual runtime memory vs. PPL for TEAL and our self-distillation.

Table 4: PPL and Downstream Task Accuracy of LLaMA-3-8B. We use TEAL as our baseline TOP-K method.

Method	PPL	MMLU	GSM8K	ARC-C	ARC-Easy	PIQA
Origin (0%)	6.0874	65.16%	50.87%	54.95%	83.96%	80.74%
TOP-K (50%)	7.7762	59.21%	32.30%	49.32%	81.40%	78.45%
TOP-K (60%)	9.0042	51.67%	17.76%	45.56%	76.98%	76.01%
TOP-K (70%)	13.6816	36.53%	3.34%	33.70%	66.60%	70.30%
TOP-K (80%)	73.1400	25.69%	1.67%	21.08%	38.89%	56.80%
Ours (50%)	6.9677	61.41%	38.89%	52.13%	81.48%	79.98%
Ours (60%)	7.7935	57.01%	28.58%	49.57%	79.46%	77.80%
Ours (70%)	9.5079	47.51%	12.89%	41.64%	74.12%	77.31%
Ours (80%)	14.6401	29.40%	2.05%	32.34%	62.30%	69.10%

serial computation and memory reads. Experimental results show that when the layer number in a cross-layer group is set to 1, the average speedup across all three devices is 10%. However, increasing the layer number to 4 results in a 120% performance improvement, as it enhances the efficiency of flash memory reads. Finally, with the addition of Dynamic Cache, our method achieves 2x, 2.3x, and 3x speedups over the baseline on the three devices, respectively.

To further understand the benefits and overhead of each technique, we conducted individual experiments for detailed analysis. As shown in Fig. 17, we evaluated the trade-offs of

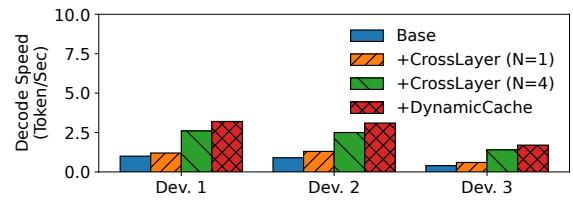


Figure 16: The decode speed improvement of LLaMA-2-7B model on three devices by each technique.

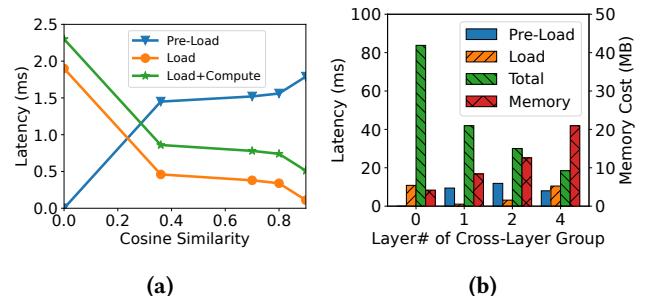


Figure 17: The performance and memory cost of cross-layer loading.

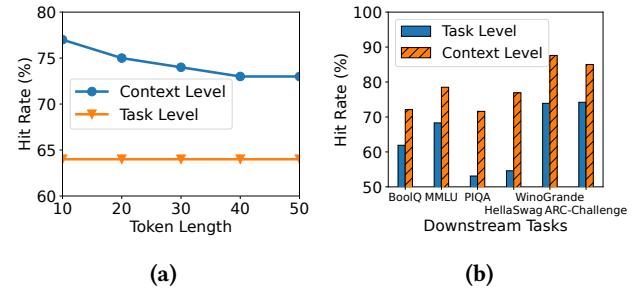


Figure 18: The performance of task-level and context-level cache.

cross-layer loading. In Fig. 17(a), we measured the loading and preloading overhead for a single layer when the layer number in a cross-layer group is set to 1, under different cosine similarity values. The results show that when cosine similarity is lower than 0.2, the preload latency is lower than the on-demand load latency. However, when cosine similarity exceeds 0.4, the on-demand load latency becomes lower than the preload latency. Since the cosine similarity of most layers is above 0.8, our cross-layer approach effectively overlaps preloading and computation, optimizing performance.

In Fig. 17(b), we evaluate an 8-layer decoder of LLaMA-2-7B, measuring preload, load, and total latency as well as memory cost under different layer numbers in a cross-layer group.

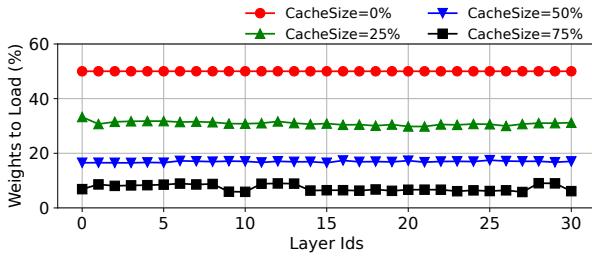


Figure 19: The rate of attention Q/K/V weights to load of LLaMA-2-7B model with 50% sparsity under various cache sizes.

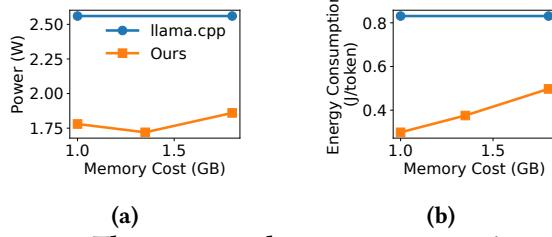


Figure 20: The power and energy consumption of ACTIVEFLOW and baseline.

When the layer number is 0, computation and flash loading occur sequentially, leading to high total latency. When the layer number increases to 1, computation begins to overlap with preloading, reducing total latency by 52%. As the layer number further increases to 4, improved preload efficiency enables a 4.1× speedup compared to the size 0 setting. However, increasing the layer number also leads to higher memory cost, introducing additional overhead. Overall, increasing the layer number in a cross-layer group effectively enhances decoding performance, while the additional memory overhead remains relatively low.

Contextual caching policy. Fig. 18 compares context-level and task-level caches. On BoolQ, when token length=10, the context-level cache achieves a 77% hit rate, 13% higher than task-level. As length increases to 40, the hit rate slightly drops to 74% but still remains 10% higher. Across downstream tasks (Fig. 18b), task-level hit rate varies between 54–74%, while context-level consistently adapts, yielding an average 12% improvement.

Cache efficiency. As shown in Fig. 19, enlarging cache size significantly reduces flash access. With 50% cache, flash operations shrink to 18% of weights, giving a 5.2× reduction in memory access compared to full loading. Larger cache further improves hit rate but also increases memory footprint; therefore, we adjust cache size dynamically based on available device memory.

Table 5: Llama-3-8B ablation studies

Method	PPL	MMLU	GSM8K	ARC-C	ARC-Easy	PIQA
Ours (50%)	6.9677	61.41%	38.89%	52.13%	81.48%	79.98%
Ours-Distill (50%)	7.4872	59.67%	32.83%	50.94%	81.82%	78.89%
Ours-STE (50%)	7.0660	60.78%	37.45%	49.91%	81.65%	79.54%
Ours (60%)	7.7935	57.01%	28.58%	49.57%	79.46%	77.80%
Ours-Distill (60%)	8.2635	55.73%	24.64%	50.68%	80.39%	77.53%
Ours-STE (60%)	8.1517	55.27%	23.58%	47.78%	78.20%	77.91%
Ours (70%)	9.5079	47.51%	12.89%	41.64%	74.12%	77.31%
Ours-Distill (70%)	9.9149	45.29%	10.77%	42.41%	73.99%	75.41%
Ours-STE (70%)	11.1969	37.72%	3.87%	35.92%	68.69%	73.18%
Ours (80%)	14.6401	29.40%	2.05%	32.34%	62.30%	69.10%
Ours-Distill (80%)	37.7404	24.44%	1.90%	20.14%	36.95%	57.67%
Ours-STE (80%)	23.2097	25.96%	1.59%	22.87%	52.74%	64.85%

7.4 Power and energy consumption

We evaluate power and energy efficiency of ACTIVEFLOW on Device 1 (Fig. 20). ACTIVEFLOW reduces average power consumption by 27.34% compared to llama.cpp due to reduced computation wait time in the overlap pipeline, and further lowers energy per token as memory cost decreases, achieving up to 53% reduction at 1.3GB memory usage.

8 Related Works

Sparsity in LLMs. Sparsity in LLMs has been the focus of many research efforts. Mirzadeh et al. [19] propose replacing the ReLU activation function in LLMs to reduce computation and weight transfer. HiRE [11] introduces high-recall approximate Top-K estimation. Prospars [22] leverages the sparsity of ReLU and gated branches in FFNs to predict model sparsity. Q-Sparse [29] trains sparse LLMs from scratch, while TEAL [15] applies magnitude-based sparsity without retraining. However, these methods either depend on ReLU-based architectures or lack mechanisms to recover accuracy under high sparsity. InfiniGen[13], FlexGen [21] and related work primarily focus on **KV cache optimization**, KV cache dominants the LLM memory usage for long context scenarios (>32K tokens), while we target **weight** memory optimization: usually determined by LLM’s weights.

Efficient LLM inference system. Several system-level efforts focus on exploiting sparsity for efficient inference. DejaVu [17] predicts contextual sparsity with lightweight algorithms, Alizadeh et al. [3] optimize inference on limited-memory devices, and PowerInfer [23] (and its extension PowerInfer-2) design CPU-GPU hybrid engines. These works primarily target *ReLU-based models* and FFN layers, often relying on heavy predictors (GB-level memory) to skip zero activations. Yet modern LLMs such as LLaMA and Mixtral adopt non-ReLU activations for accuracy [27], limiting the applicability of these methods. LLM-in-Flash [1] streams weights from flash with fine-grained prefetching to reduce

DRAM usage, but its efficiency is limited by flash bandwidth and latency, especially for compute-intensive layers.

Our distinction. ACTIVEFLOW eliminates the ReLU dependency and predictor overhead by targeting *all weights* (Attention and FFN) in modern non-ReLU LLMs. It targets *all weights* (both Attention and FFN) and eliminates the need for predictors. It introduces (1) cross-layer active weight preloading, generalizing cross-layer similarity, and (2) sparsity-aware self-distillation to recover accuracy under high sparsity. Combined with an LFU-based cache driven by activation statistics, our approach consistently achieves higher hit rates (e.g., >70% vs. ~55%) and ensures strict memory budgets, enabling reliable edge deployment.

Static pruning techniques. Static pruning and quantization are established methods for compressing large language models (e.g., CFSP [33], DB-LLM [6], and RIA [5]). Although effective at reducing model size and computation, these static approaches require offline processing of model weights, which limits flexibility for dynamic tasks. Our system, ACTIVEFLOW, is not only compatible with these static techniques but also uniquely supports dynamic processing to address this challenge.

9 Conclusion

This paper proposes the first LLM inference system on mobile devices that supports adaptive DRAM usage, in order to scale up the deployable model size. It is based on the idea of active weight swapping between DRAM and flash, integrating three novel techniques: cross-layer active weight preloading, sparsity-aware self-distillation, and active weight swapping pipeline. It achieves the inference performance-cost Pareto frontier compared to other efficiency optimization methods. This paper breaks the DRAM limitation for LLM deployment, opening up the new opportunity of server-level LLMs deployment on mobile devices.

References

- [1] Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. arXiv:2312.11514 [cs.CL] <https://arxiv.org/abs/2312.11514>
- [2] Keivan Alizadeh, Seyed-Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C. del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. LLM in a flash: Efficient Large Language Model Inference with Limited Memory. CoRR abs/2312.11514 (2023). <https://doi.org/10.48550/ARXIV.2312.11514> arXiv:2312.11514
- [3] Keivan Alizadeh, Seyed-Iman Mirzadeh, Dmitry Belenko, S Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. Llm in a flash: Efficient large language model inference with limited memory. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 12562–12584.
- [4] Apple. [n. d.]. Introducing Apple’s On-Device and Server Foundation Models. <https://machinelearning.apple.com/research/introducing-apple-foundation-models>.
- [5] Anmol Biswas, Raghav Singhal, Sivakumar Elangovan, Shreyas Sabnis, and Udayan Ganguly. 2025. Regularization-based Framework for Quantization-, Fault- and Variability-Aware Training. arXiv:2503.01297 [cs.LG] <https://arxiv.org/abs/2503.01297>
- [6] Hong Chen, Chengtao Lv, Liang Ding, Haotong Qin, Xiabin Zhou, Yifu Ding, Xuebo Liu, Min Zhang, Jinyang Guo, Xianglong Liu, and Dacheng Tao. 2024. DB-LLM: Accurate Dual-Binarization for Efficient LLMs. arXiv:2402.11960 [cs.LG] <https://arxiv.org/abs/2402.11960>
- [7] Dayou Du, Yijia Zhang, Shijie Cao, Jiaqi Guo, Ting Cao, Xiaowen Chu, and Ningyi Xu. 2024. BitDistiller: Unleashing the Potential of Sub-4-Bit LLMs via Self-Distillation. arXiv:2402.10631 [cs.CL]
- [8] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can be Accurately Pruned in One-Shot. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 10323–10337. <https://proceedings.mlr.press/v202/frantar23a.html>
- [9] Yizhao Gao, Zhichen Zeng, Dayou Du, Shijie Cao, Hayden Kwok-Hay So, Ting Cao, Fan Yang, and Mao Yang. 2024. SeerAttention: Learning Intrinsic Sparse Attention in Your LLMs. CoRR abs/2410.13276 (2024). <https://doi.org/10.48550/ARXIV.2410.13276> arXiv:2410.13276
- [10] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of Experts. arXiv:2401.04088 [cs.LG] <https://arxiv.org/abs/2401.04088>
- [11] Yashas Samaga B L, Varun Yerram, Chong You, Srinadh Bhojanapalli, Sanjiv Kumar, Prateek Jain, and Praneeth Netrapalli. 2024. HiRE: High Recall Approximate Top- k Estimation for Efficient LLM Inference. arXiv:2402.09360 [cs.LG] <https://arxiv.org/abs/2402.09360>
- [12] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. CoRR abs/2406.19707 (2024). <https://doi.org/10.48550/ARXIV.2406.19707> arXiv:2406.19707
- [13] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. 2024. InfiniGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. arXiv:2406.19707 [cs.LG] <https://arxiv.org/abs/2406.19707>
- [14] Xiangyu Li, Yuanchun Li, Yuanzhe Li, Ting Cao, and Yunxin Liu. 2024. FlexNN: Efficient and Adaptive DNN Inference on Memory-Constrained Edge Devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2024, Washington D.C., DC, USA, November 18-22, 2024*, Weisong Shi, Deepak Ganesh, and Nicholas D. Lane (Eds.). ACM, 709–723. <https://doi.org/10.1145/3636534.3649391>
- [15] James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. 2025. Training-Free Activation Sparsity in Large Language Models. arXiv:2408.14690 [cs.CL] <https://arxiv.org/abs/2408.14690>
- [16] James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. 2025. Training-Free Activation Sparsity in Large Language Models. arXiv:2408.14690 [cs.CL] <https://arxiv.org/abs/2408.14690>
- [17] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023. Deja Vu: Contextual Sparsity for Efficient

- LLMs at Inference Time. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 22137–22176. <https://proceedings.mlr.press/v202/liu23am.html>
- [18] Microsoft. [n. d.]. Phi Silica, small but mighty on-device SLM. <https://blogs.windows.com/windowsexperience/2024/12/06/phi-silica-small-but-mighty-on-device-slm/>.
- [19] 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:2310.04564 [cs.LG] <https://arxiv.org/abs/2310.04564>
- [20] Yuzhang Shang, Zhihang Yuan, Qiang Wu, and Zhen Dong. 2023. PB-LLM: Partially Binarized Large Language Models. arXiv:2310.00034 [cs.LG] <https://arxiv.org/abs/2310.00034>
- [21] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU. arXiv:2303.06865 [cs.LG] <https://arxiv.org/abs/2303.06865>
- [22] Chenyang Song, Xu Han, Zhengyan Zhang, Shengding Hu, Xiyu Shi, Kuai Li, Chen Chen, Zhiyuan Liu, Guangli Li, Tao Yang, and Maosong Sun. 2024. ProSparse: Introducing and Enhancing Intrinsic Activation Sparsity within Large Language Models. CoRR abs/2402.13516 (2024). <https://doi.org/10.48550/ARXIV.2402.13516> arXiv:2402.13516
- [23] 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 (Austin, TX, USA) (SOSP '24)*. Association for Computing Machinery, New York, NY, USA, 590–606. <https://doi.org/10.1145/3694715.3695964>
- [24] Yixin Song, Haotong Xie, Zhengyan Zhang, Bo Wen, Li Ma, Zeyu Mi, and Haibo Chen. 2024. Turbo Sparse: Achieving LLM SOTA Performance with Minimal Activated Parameters. arXiv:2406.05955 [cs.LG] <https://arxiv.org/abs/2406.05955>
- [25] 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, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net. <https://openreview.net/forum?id=PxoFut3dWW>
- [26] Gemini Team. 2024. Gemini: A Family of Highly Capable Multimodal Models. <https://doi.org/10.48550/arXiv.2312.11805> arXiv:2312.11805
- [27] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenjin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Sagar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yunling Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL]
- [28] Hongyu Wang, Shuming Ma, Ruiping Wang, and Furu Wei. 2024. Q-Sparse: All Large Language Models can be Fully Sparsely-Activated. CoRR abs/2407.10969 (2024). <https://doi.org/10.48550/ARXIV.2407.10969> arXiv:2407.10969
- [29] Hongyu Wang, Shuming Ma, Ruiping Wang, and Furu Wei. 2024. Q-Sparse: All Large Language Models can be Fully Sparsely-Activated. arXiv:2407.10969 [cs.CL] <https://arxiv.org/abs/2407.10969>
- [30] Jinheung Wang, Hansong Zhou, Ting Song, Shijie Cao, Yan Xia, Ting Cao, Jianyu Wei, Shuming Ma, Hongyu Wang, and Furu Wei. 2025. Bitnet.cpp: Efficient Edge Inference for Ternary LLMs. arXiv:2502.11880 [cs.LG] <https://arxiv.org/abs/2502.11880>
- [31] Tuowei Wang, Ruwen Fan, Minxing Huang, Zixu Hao, Kun Li, Ting Cao, Youyou Lu, Yaoxue Zhang, and Ju Ren. 2024. Ripple: Accelerating LLM Inference on Smartphones with Correlation-Aware Neuron Management. arXiv:2410.19274 [cs.LG] <https://arxiv.org/abs/2410.19274>
- [32] Yuxin Wang, Minghua Ma, Zekun Wang, Jingchang Chen, Huiming Fan, Liping Shan, Qing Yang, Dongliang Xu, Ming Liu, and Bing Qin. 2024. CFSP: An Efficient Structured Pruning Framework for LLMs with Coarse-to-Fine Activation Information. arXiv:2409.13199 [cs.CL] <https://arxiv.org/abs/2409.13199>
- [33] Yuxin Wang, MingHua Ma, Zekun Wang, Jingchang Chen, Shan Liping, Qing Yang, Dongliang Xu, Ming Liu, and Bing Qin. 2025. CFSP: An Efficient Structured Pruning Framework for LLMs with Coarse-to-Fine Activation Information. In *Proceedings of the 31st International Conference on Computational Linguistics*, Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schock-aert (Eds.). Association for Computational Linguistics, Abu Dhabi, UAE, 9311–9328. <https://aclanthology.org/2025.coling-main.626/>
- [34] Jianyu Wei, Shijie Cao, Ting Cao, Lingxiao Ma, Lei Wang, Yanyong Zhang, and Mao Yang. 2024. T-MAC: CPU Renaissance via Table Lookup for Low-Bit LLM Deployment on Edge. arXiv:2407.00088 [cs]
- [35] Apple Xcode. 2024. Reducing your app’s memory use. <https://developer.apple.com/documentation/xcode/reducing-your-app-s-memory-use>
- [36] Zhenliang Xue, Yixin Song, Zeyu Mi, Le Chen, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. arXiv:2406.06282 [cs.LG] <https://arxiv.org/abs/2406.06282>
- [37] Kai Yi and Peter Richtárik. 2025. Symmetric Pruning of Large Language Models. arXiv:2501.18980 [cs.LG] <https://arxiv.org/abs/2501.18980>
- [38] Rongjie Yi, Ting Cao, Ao Zhou, Xiao Ma, Shangguang Wang, and Mengwei Xu. 2023. Boosting DNN Cold Inference on Edge Devices. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services, MobiSys 2023, Helsinki, Finland, June 18-22, 2023*, Petteri Nurmi, Pan Hui, Ardalan Amiri Sani, and Yunxin Liu (Eds.). ACM, 516–529. <https://doi.org/10.1145/3581791.3596842>
- [39] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. 2025. Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, 23078–23097. <https://aclanthology.org/2025.acl-long.1126>