arXiv:2312.11514v1 [cs.CL] 12 Dec 2023

LLM *in a flash*: Efficient Large Language Model Inference with Limited Memory

Keivan Alizadeh[*], Iman Mirzadeh[†], Dmitry Belenko[‡], Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, Mehrdad Farajtabar[§]
Apple

Abstract

Large language models (LLMs) are central to modern natural language processing, delivering exceptional performance in various tasks. However, their intensive computational and memory requirements present challenges, especially for devices with limited DRAM capacity. This paper tackles the challenge of efficiently running LLMs that exceed the available DRAM capacity by storing the model parameters on flash memory but bringing them on demand to DRAM. Our method involves constructing an inference cost model that harmonizes with the flash memory behavior, guiding us to optimize in two critical areas: reducing the volume of data transferred from flash and reading data in larger, more contiguous chunks. Within this flash memory-informed framework, we introduce two principal techniques. First, "windowing" strategically reduces data transfer by reusing previously activated neurons, and second, "row-column bundling", tailored to the sequential data access strengths of flash memory, increases the size of data chunks read from flash memory. These methods collectively enable running models up to twice the size of the available DRAM, with a 4-5x and 20-25x increase in inference speed compared to naive loading approaches in CPU and GPU, respectively. Our integration of sparsity awareness, context-adaptive loading, and a hardware-oriented design paves the way for effective inference of LLMs on devices with limited memory.

1 Introduction

In recent years, large language models (LLMs), such as GPT-3 (Brown et al., 2020), OPT (Zhang et al., 2022b), and PaLM (Chowdhery et al., 2022), have demonstrated strong performance across a wide range of natural language tasks. However, the unprecedented capabilities of these models come with substantial computational and memory requirements for inference. LLMs can contain hundreds of billions or even trillions of parameters, making it challenging to load and run them efficiently, especially on resource-constrained devices.

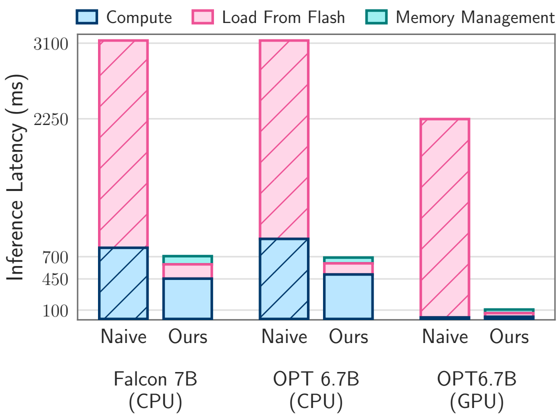
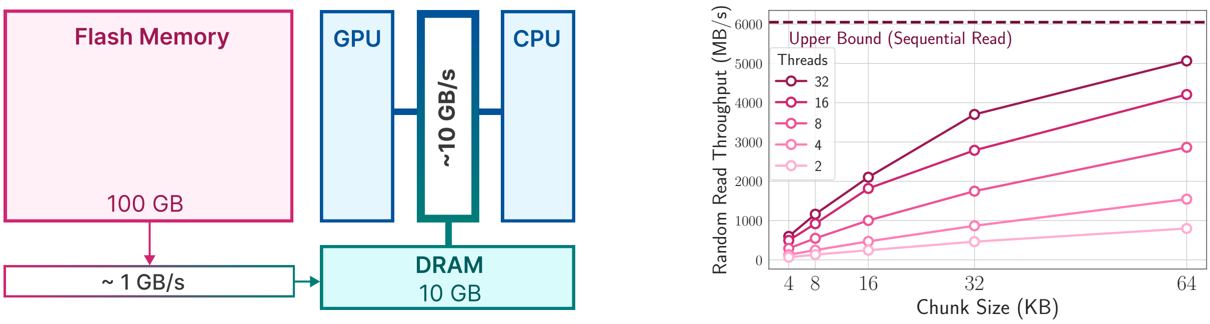


Figure 1: Inference latency of 1 token when half the memory of the model is available.

Currently, the standard approach is to load the entire model into DRAM for inference (Rajbhandari et al., 2021; Aminabadi et al., 2022). However, this severely limits the maximum model size that can be run. For example, a 7 billion parameter model requires over 14GB of memory just to load the parameters in half-precision floating point format, exceeding the capabilities of most edge devices.

To address this limitation, we propose to store the model parameters on flash memory, which is at least an order of magnitude larger than DRAM. Then, during inference, we directly and cleverly load the required parameters from the flash memory, avoiding the need to fit the entire model in DRAM. Our methodology is built on the top of recent works that have shown LLMs exhibit a high degree of sparsity in the FeedForward Network (FFN) layers, with models like OPT (Zhang et al.,

[*]Primary Author: kalizadehvahid@apple.com
[†]Major Contribution: imirzadeh@apple.com
[‡]Major Contribution: d_belenko@apple.com
[§]Senior Author: farajtabar@apple.com



(a) Bandwidth in a unified memory architecture

(b) Random read throughput of flash memory

Figure 2: **(a)** Flash memory offers significantly higher capacity but suffers from much lower bandwidth compared to DRAM and CPU/GPU caches and registers. **(b)** The throughput for random reads in flash memory increases with the size of sequential chunks and the number of threads.

2022b), Falcon (Almazrouei et al., 2023), exhibiting more than 90% sparsity (Mirzadeh et al., 2023; Liu et al., 2023b). We exploit this sparsity to selectively load only parameters from flash memory that either have non-zero input or are predicted to have non-zero output. Specifically, we discuss a hardware-inspired cost model that includes flash memory, DRAM, and computing cores (CPU or GPU). Then, we introduce two complementary techniques to minimize data transfer and maximize flash memory throughput:

- Windowing: We load parameters for only the past few tokens, reusing activations from recently computed tokens. This sliding window approach reduces the number of IO requests to load weights.

- Row-column bundling: We store a concatenated row and column of the up-projection and down-projection layers to read bigger contiguous chunks from flash memory. This increases throughput by reading larger chunks.

To further minimize the number of weights to be transferred from flash memory to DRAM, we also employ methods to predict FFN sparsity and avoid loading zeroed-out parameters, akin to approaches documented in Deja Vu (Li and Lu, 2023). Together, windowing and sparsity prediction allow us to load only 2% of the FFN layer from flash for each inference query. We also propose a static memory preallocation to minimize transfers within DRAM and reduce inference latency. Our load from flash cost model captures the tradeoff between loading less data and reading bigger chunks. Optimizing this cost model and selectively loading parameters on demand yields flash loading strategies that can run models 2x larger than the device's DRAM capacity and speed up inference by 4-5x and 20-25x compared to naive implementation in CPU and GPU, respectively.

2 Flash Memory & LLM Inference

In this section, we explore the characteristics of memory storage systems (e.g., flash, DRAM), and their implications for large language model (LLM) inference. Our aim is to elucidate the challenges and hardware-specific considerations essential for algorithm design, particularly in optimizing inference when working with flash memory.

2.1 Bandwidth and Energy Constraints

While a modern NAND flash memories offers high bandwidth and low latency, it falls short of the performance levels of DRAM (Dynamic Random-Access Memory), especially in memory-constrained systems. Figure 2a illustrates these differences. A naive inference implementation that relies on NAND flash memory might necessitate reloading the entire model for each forward pass. This process is not only time-consuming, often taking seconds for even compressed models, but it also consumes more energy than transferring data from DRAM to the CPU or GPU's internal memory.

In scenarios where DRAM is abundant, the cost of loading data is somewhat mitigated, as the model can reside in DRAM. However, the initial loading of the model still incurs a penalty, particularly in situations requiring rapid response times for the first token. Our approach, leveraging activation sparsity in LLMs, addresses these challenges by enabling selective reading of model weights, thereby reducing both time and power costs.

2.2 Read Throughput

Flash memory systems perform optimally with large sequential reads. For instance, benchmarks on an Apple MacBook Pro M2 with 2TB flash demonstrate speeds exceeding 6GiB/s for a 1GiB linear read of an uncached file. However, this high bandwidth is not replicated for smaller, random reads due to the inherent multi-phase nature of these reads, encompassing the operating system, drivers, interrupt handling, and the flash controller, among others. Each phase introduces latency, disproportionately affecting smaller reads.

To circumvent these limitations, we advocate two primary strategies, which can be employed jointly. The first involves reading larger chunks of data. Although throughput growth is not linear (larger chunks take longer to transfer), the latency for the initial byte becomes a smaller fraction of the total request time, resulting in more efficient data reading. This principle is depicted in Figure 2b. Perhaps a counterintuitive yet interesting observation is that in some scenarios, it will be faster to read more than needed (but in larger chunks) and then discard, than only reading necessary parts but in smaller chunks. The second strategy leverages parallelized reads, utilizing the inherent parallelism within storage stacks and flash controllers. Our results indicate that throughputs appropriate for sparse LLM inference are achievable on standard hardware using 32KiB or larger random reads across multiple threads.

Crucial to maximizing throughput is the way weights are stored, as a layout that enhances the average chunk length can significantly boost bandwidth. In some cases, it might be beneficial to read and subsequently discard excess data, rather than splitting the data into smaller, less efficient chunks.

Motivated by the challenges described in this section, in section 3, we propose methods to optimize data transfer volume and enhance read throughput to significantly enhance inference speeds.

3 Load From Flash

This section addresses the challenge of conducting inference on devices where the available computational memory is substantially smaller than the size of the model. This necessitates storing the full model weights in flash memory. Our primary metric for evaluating various flash loading strategies is latency, dissected into three distinct components: the I/O cost of loading from flash, the overhead of managing memory with newly loaded data, and the compute cost for inference operations.

Our proposed solutions for reducing latency under memory constraints are categorized into three strategic areas, each targeting a specific aspect of the latency:

- **Reducing Data Load**: Aiming to decrease latency associated with flash I/O operations by loading less data[1].
- **Optimizing Data Chunk Size**: Enhancing flash throughput by increasing the size of data chunks loaded, thereby mitigating latency.
- **Efficient Management of Loaded Data**: Streamlining the management of data once it is loaded into memory to minimize overhead.

It is important to note that our focus is not on the compute aspect of the process, as it is orthogonal to the core concerns of our work. This delineation allows us to concentrate on optimizing flash memory interactions and memory management to achieve efficient inference on memory-constrained devices.

Finally, we will elaborate on the implementation of these strategies in subsequent sections.

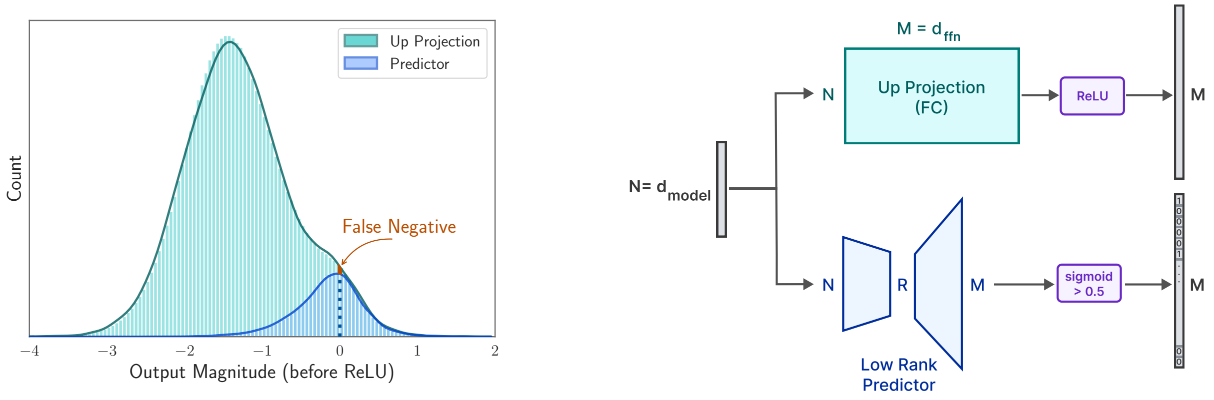
3.1 Reducing Data Transfer

Our methodology leverages the inherent sparsity found in Feed-Forward Network (FFN) models, as documented in preceding research. The OPT 6.7B model, for instance, exhibits a notable 97% sparsity within its FFN layer. Similarly, the Falcon 7B model has been adapted through fine-tuning, which involves swapping their activation functions to ReLU, resulting in 95% sparsity while being almost similar in accuracy (Mirzadeh et al., 2023). In light of this information, our approach involves the iterative transfer of only the essential, non-sparse data from flash memory to DRAM for processing during inference.

It's notable that, we employ the 7B models as a practical example to elucidate our approach, but our findings are adaptable and can be extrapolated to both larger and smaller scale models with ease.

Selective Persistence Strategy. We opt to retain the embeddings and matrices within the attention mechanism of the transformer constantly

[1] It is notable that, by *data* we mean weights of the neural network. However, our developed techniques can be easily generalized to other data types transferred and used for LLM inference, such as activations or KV cache, as suggested by (Sheng et al., 2023).



(a) predictor vs relu

(b) low rank predictor

Figure 3: **(a)** Preactivations of tokens in one sequence in OPT 6.7B. The blue graph shows preactivation of elements that predictor detected positive while the green graph is for up projection. As it can be seen most of the False Positives are close to 0 and False Negatives constitute a small portion of the elements. **(b)** A small low rank predictor finds out which intermediate neurons are going to be activated instead of running heavy up projection.

in RAM. For the Feed-Forward Network (FFN) portions, only the non-sparse segments are dynamically loaded into DRAM as needed. Storing attention weights, which constitute approximately one-third of the model's size, in memory, allows for more efficient computation and quicker access, thereby enhancing inference performance without the need for full model loading.

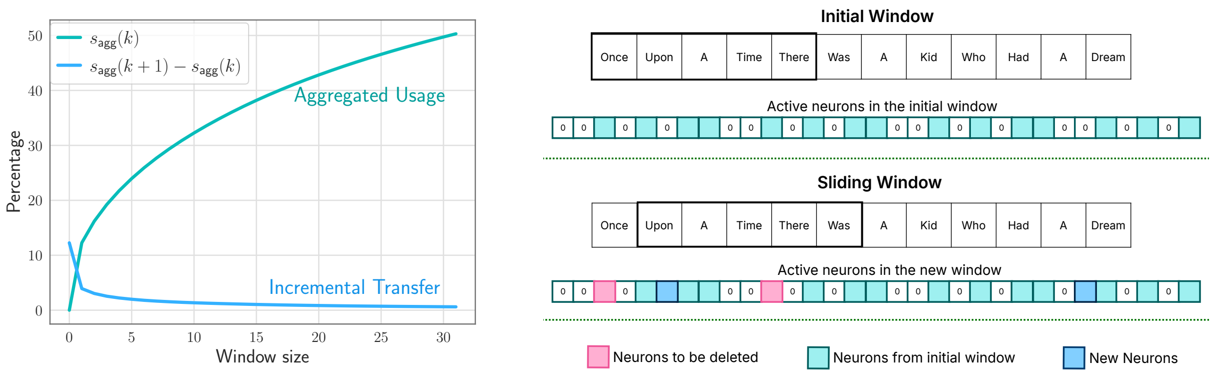
Anticipating ReLU Sparsity. The ReLU activation function naturally induces over 90% sparsity in the FFN's intermediate outputs, which reduces the memory footprint for subsequent layers that utilize these sparse outputs. However, the preceding layer, namely the up project for OPT and Falcon, must be fully present in memory. To circumvent loading the entire up project matrix, we follow Liu et al. (2023b) and employ a low-rank predictor to identify the zeroed elements post-ReLU (see Figure 3b). In contrast to their work, our predictor needs only the output of the current layer's attention module, and not the previous layer's FFN module. We have observed that, postponing the prediction to current layer is sufficient for a hardware aware weight loading algorithm design, but, leads to more accurate outcome due to deferred inputs. We thereby only load elements indicated by the predictor.

Neuron Data Management via Sliding Window Technique. In our study, we define an *active neuron* as one that yields a positive output in our predictive model. Our approach focuses on managing neuron data by employing a *Sliding Window Technique*. This methodology entails maintaining neuron data only for a recent subset of input tokens in the memory. The key aspect of this technique is the selective loading of neuron data that differs between the current input token and its immediate predecessors. This strategy allows for efficient memory utilization, as it frees up memory resources previously allocated to neuron data from older tokens that are no longer within the sliding window (as depicted in Figure 4b).

Table 1: Using predictors doesn't change the accuracy of zero-shot metrics significantly as predictor of each layer accurately identifies sparsity

Zero-Shot Task	OPT 6.7B	with Predictor
Arc Easy	66.1	66.2
Arc Challenge	30.6	30.6
HellaSwag	50.3	49.8

From a mathematical standpoint, let $s_{\text{agg}}(k)$ denote the cumulative use of neuron data across a sequence of k input tokens. Our memory architecture is designed to store an average of $s_{\text{agg}}(k)$ in Dynamic Random-Access Memory (DRAM). As we process each new token, the incremental neuron data, which is mathematically represented as $s_{\text{agg}}(k+1) - s_{\text{agg}}(k)$, is loaded from flash memory into DRAM. This practice is grounded in the observed trend of decreasing aggregated neuron usage over time. Consequently, larger values of k result in a lesser volume of data being loaded for each new token. (refer to Figure 4a) This reduction in data loading is counterbalanced by the memory cost associated with storing $s_{\text{agg}}(k)$. In determining the size of the sliding window, the aim is to maximize it within the constraints imposed by the available memory capacity.



(a) aggregated neuron use

(b) sliding window

Figure 4: **(a)** Aggregated neuron use of the tenth layer of Falcon 7B, as it can be seen the slop of aggregated neuron use is decreasing. Other layers exhibit the same pattern. **(b)** Instead of deleting neurons that brought to DRAM we keep the active neurons of past 5 tokens: when the new token "Was" is being processed only a few amount of data needs to be changed.

3.2 Improving Transfer Throughput with Increased Chunk Sizes

To increase data throughput from flash memory, it is crucial to read data in more substantial chunks. In this section, we detail the strategy we have employed to augment the chunk sizes for more efficient flash memory reads.

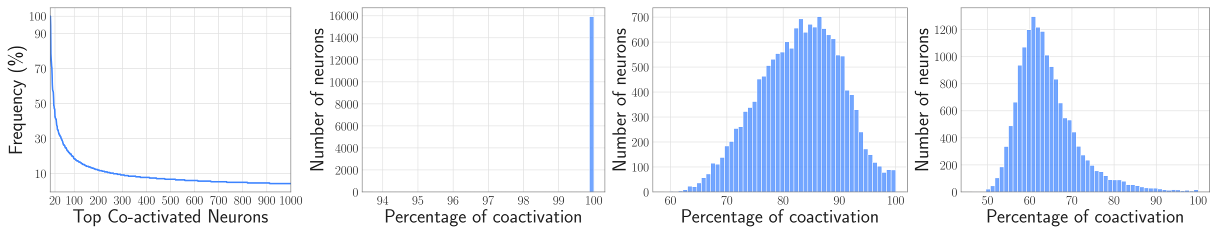
Bundling Columns and Rows. For OPT and Falcon models, the usage of the ith column from the upward projection and the ith row from the downward projection coincides with the activation of the ith intermediate neuron. Consequently, by storing these corresponding columns and rows together in flash memory, we can consolidate the data into larger chunks for reading. Refer to Figure 7 for an illustration of this bundling approach. If each element of weights of the network is stored in *num_bytes* such bundling doubles the chunk size from $d_{model} \times$*num_bytes* to $2d_{model} \times$*num_bytes* as shown in Figure 7. Our analysis and experiment show this increases the throughput of the model.

Bundling Based on Co-activation. We had a conjecture that neurons may be highly correlated in their activity patterns, which may enable further bundling. To verify this we calculated the activations of neurons over C4 validation dataset. For each neuron the coactivation of that neuron with other ones forms a power law distribution as depicted in Figure 5a. Now, let's call the neuron that coactivates with a neuron the most *closest friend*. Indeed, the closest friend of each neuron coactivates with it very often. As Figure 5b demonstrates, it is interesting to see each neuron and its closest friend coactivate with each other at least 95% of the times. The graphs for the 4th closest friend and 8th closest friend are also drawn. Based on this information we decided to put a bundle of each neuron and its closest friend in the flash memory; whenever a neuron is predicted to be active we'll bring its closes friend too. Unfortunately, this resulted in loading highly active neurons multiple times and the bundling worked against our original intention. It means, the neurons that are very active are 'closest friend' of almost everyone. We intentionally present this negative result, as we believe it may lead to interesting future research studies on how to effectively bundle the neurons and how to leverage it for efficient inference.

3.3 Optimized Data Management in DRAM

Although data transfer within DRAM is more efficient compared to accessing flash memory, it still incurs a non-negligible cost. When introducing data for new neurons, reallocating the matrix and appending new matrices can lead to significant overhead due to the need for rewriting existing neurons data in DRAM. This is particularly costly when a substantial portion (approximately 25%) of the Feed-Forward Networks (FFNs) in DRAM needs to be rewritten. To address this issue, we adopt an alternative memory management strategy. This involves the preallocation of all necessary memory and the establishment of a corresponding data structure for efficient management. The data structure comprises elements such as `pointers`, `matrix`, `bias`, `num_used`, and `last_k_active` shown in Figure 6.

Each row in the `matrix` represents the concate-



(a) coactivation intensity (b) Closest friend (c) 4th closest friend (d) 8th closest friend

Figure 5: **(a)** coactivation of the neurons of 10th layer with one random neuron in the same layer shows a few neurons are highly coactivated with it and can be grouped **(b)** closest friend of a neuron is the one that gets coactivated with it the most, coactivation of closest friend of every neuron in opt125m shows closest friend of each neuron almost always gets coactivated with it. **(c)** The 3rd closest friend gets coactivatd with each neuron 86% of the time in average **(d)** The 7th closest friend seems to be less relevant and doesnt coactivate with the neuron very often

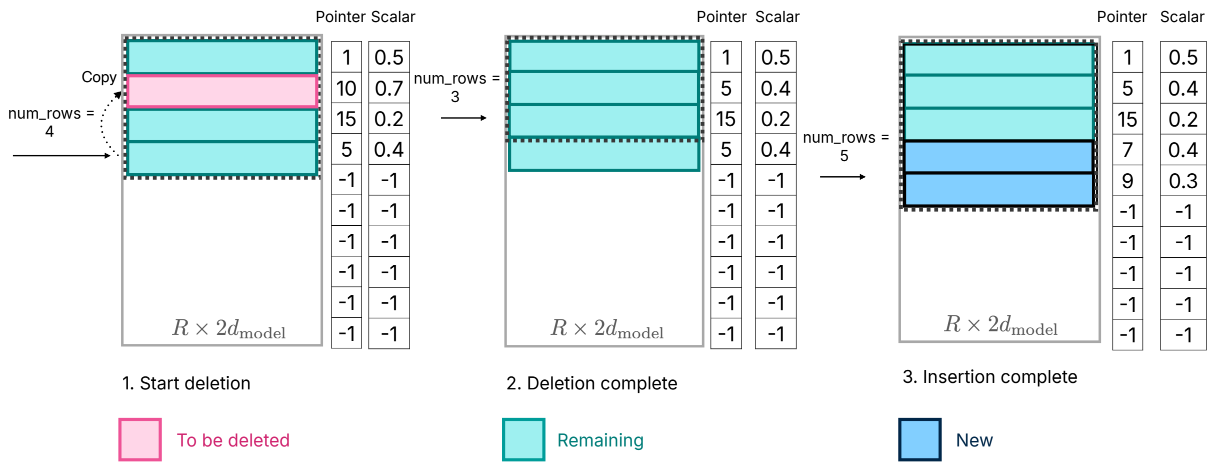


Figure 6: Memory management, first we copy last elements to deleting neurons to maintain a consecutive block of memory then the required ones are stack to the end, this prevents from copying whole data multiple times

nated row of the 'up project' and the column of the 'down project' of a neuron. The `pointer` vector indicates the original neuron index corresponding to each row in the `matrix`. The bias for the 'up project' in the original model is represented in the corresponding `bias` element. The `num_used` parameter tracks the number of rows currently utilized in the `matrix`, initially set to zero. The `matrix` for the ith layer is pre-allocated with a size of $\texttt{Req}_i \times 2d_{\text{model}}$, where $\texttt{Req}_i$ denotes the maximum number of neurons required for the specified window size in a subset of C4 validation set. By allocating a sufficient amount of memory for each layer in advance, we minimize the need for frequent reallocation. Finally, the `last_k_active` component identifies the neurons from the original model that were most recently activated using the last k tokens.

The following operations are done during inference as depicted in Figure 6.

1. **Deleting Neurons:** Neurons that are no longer required are identified efficiently in linear time, utilizing the `last_k_active` data and the current prediction. The `matrix`, `pointer`, and `scalars` of these redundant neurons are replaced with the most recent elements, and their count is subtracted from `num_rows`. For $O(c)$ neurons to be deleted, a memory rewrite of the order $O(c \times d_{\text{model}})$ is required.

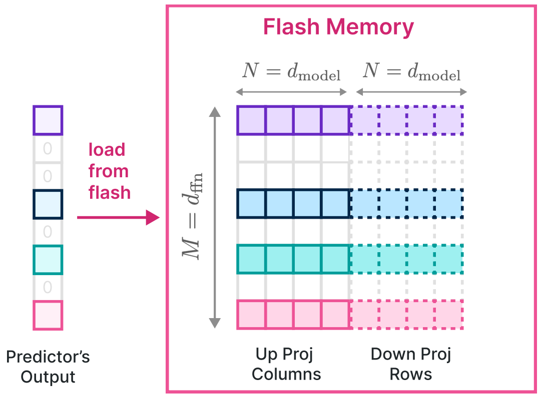


Figure 7: By bundling columns of up project and rows of down project in OPT 6.7B we will load 2x chunks instead of reading columns or rows separately.

2. **Bringing in New Neurons:** Necessary neuron data is retrieved from flash memory. The corresponding pointers and scalars are read from DRAM, and these rows are then inserted into the matrix, extending from `num_row` to `num_row + num_new`. This approach eliminates the need for reallocating memory in DRAM and copying existing data, reudcing inference latency.

3. **Inference Process:** For the inference operation, the first half of the `matrix[:num_rows,:d_model]` is used as the 'up project', and the transposed second half, `matrix[:num_rows,d_model:].transpose()`, serves as the 'down project'. This configuration is possible because the order of neurons in the intermediate output of the feed-forward layer does not alter the final output, allowing for a streamlined inference process.

These steps collectively ensure efficient memory management during inference, optimizing the neural network's performance and resource utilization.

4 Results

Experimental Setup: Our experiment is designed to optimize inference efficiency on personal devices. To this end, we process sequences individually, running only one sequence at a time. This approach allows us to allocate a specific portion of DRAM for the Key-Value (KV) cache while primarily focusing on the model size. This strategy is particularly effective when dealing with only one sequence/query at a time.[2]

[2]For OPT 6.7 B model with context length 2048 KV-cache requires $2048 \times 2d_{\text{model}}$ elements which is only 8% of model size. Also the KV-cache itself can be held in flash memory.

For the implementation of our inference process, we utilize the HuggingFace's transformers and KV caching. This setup is tested under the condition where approximately half of the model size is available in DRAM. We select this amount as a showcase of the idea of hosting the LLM in flash. With a different level of sparsity or employing quantization, one can work with smaller available DRAM capacity as well. Such a configuration demonstrates the practicality of executing inference with lower memory footprints.

Hardware Configuration. Our models are evaluated using two distinct hardware setups. The first setup includes an Apple M1 Max with a 1TB solid-state drive (SSD) for flash memory. In this configuration, computations are performed on the CPU, and the models are maintained in a 32-bit format. The second setup involves a Linux machine equipped with a 24 GB NVIDIA GeForce RTX 4090 graphics card. For this machine, computations are GPU-based, and models are run in the bfloat16 format. For both setups, we operate under the assumption that almost half of the total available memory (DRAM plus GPU memory) is allocated for model computations.

Models. We use OPT 6.7B (Zhang et al., 2022b) and a sparsified Falcon 7B (Mirzadeh et al., 2023) model for our evaluations.

Baselines. For methods not employing sparsity or weight sharing, at least half of the model must be transferred from flash memory during the forward pass. This necessity arises because, initially, only half of the model is available in DRAM, but as the forward pass progresses, the entire model capacity is utilized. Consequently, any data not present at the start must be transferred at least once. Thus, the most efficient theoretical baseline involves loading half of the model size from the flash memory into DRAM. This optimal I/O scenario serves as our primary baseline. Comparative methods, such as FlexGen (Sheng et al., 2023) and Petals (Borzunov et al., 2023), are also constrained by the limited available DRAM or GPU memory, and therefore cannot surpass this theoretical I/O efficiency.

Flash memory Data Loading Implementation. To optimize data loading from flash memory, our system employs a 32-thread reading process. This multithreading approach is specifically designed to enhance data retrieval efficiency, allowing for simultaneous access to multiple data segments (Figure 2b).

Caching Considerations for Data Loading

from Flash Memory. When data is read from flash memory, the operating system typically caches these pages, anticipating future reuse. However, this caching mechanism consumes additional memory in DRAM beyond what is allocated for the model. To accurately assess the real throughput of flash memory under limited DRAM conditions, benchmarks should be conducted without relying on caching.

For the purpose of our hardware benchmarking in this study, we deliberately constrain our NVMe throughput measurements. On macOS and iOS, we employ the `F_NOCACHE` flag with the `fcntl()` function, while on Linux, we use `DirectIO`. Additionally, on macOS, we clear any resident buffers before initiating the benchmark using the `purge` command. This approach provides a conservative estimate of throughput in scenarios where no caching is permitted. It's worth noting that these figures can improve if either the inference code or the operating system caches some part of the weights.

While OS-level buffer caching is advantageous for general applications with high cache hit rates, it lacks fine-grained control over cache usage per process or buffer eviction at the application level. In the context of on-device memory constraints and large model sizes, this could lead to ineffective caching cycles and buffer eviction, leading to minimal gains and potential issues with memory allocation and Translation Lookaside Buffer (TLB) churn.

4.1 Results for OPT 6.7B Model

This section presents the outcomes for the OPT 6.7B model, specifically under conditions where the memory allocated for the model in DRAM is approximately half of its baseline requirement.

Predictors. For the initial 28 layers of the OPT 6.7B model, we train predictors with a rank of $r = 128$. To reduce the occurrence of false negatives, the final four layers employ predictors with a higher rank of $r = 1024$. These predictors achieve an average of 5% false negatives and 7% false positives in the OPT 6.7B model. As depicted in Figure 3a, our predictor accurately identifies most activated neurons, while, occasionally misidentifying inactive ones with values near zero. Notably, these false negatives, being close to zero, do not significantly alter the final output when they are excluded. Furthermore, as demonstrated in Table 1, this level of prediction accuracy does not adversely affect the model's performance in 0-shot tasks.

Windowing in the OPT 6.7B Model. Utilizing a windowing method with $k = 5$ in the OPT 6.7B model significantly reduces the necessity for fresh data loading. Using active neurons of predictor would require about 10% of the DRAM memory capacity in average; however, with our method, it drops to 2.4%. This process involves reserving DRAM memory for a window of the past 5 tokens, which, in turn, increases the DRAM requirement for the Feed Forward Network (FFN) to 24%.

The overall memory retained in DRAM for the model comprises several components: Embeddings, the Attention Model, the Predictor, and the Loaded Feed Forward layer. The Predictor accounts for 1.25% of the model size, while Embeddings constitute 3%. The Attention Model's weights make up 32.3%, and the FFN occupies 15.5% (calculated as 0.24×64.62). Summing these up, the total DRAM memory usage amounts to 52.1% of the model's size.

Latency Analysis: Using a window size of 5, each token requires access to 2.4% of the Feed Forward Network (FFN) neurons. For a 32-bit model, the data chunk size per read is $2d_{\text{model}} \times 4$ bytes $= 32$ KiB, as it involves concatenated rows and columns. On an M1 Max, this results in a latency of $125ms$ per token for loading from flash and $65ms$ for memory management (involving neuron deletion and addition). Thus, the total memory-related latency is less than $190ms$ per token (refer to Figure 1). In contrast, the baseline approach, which requires loading 13.4GB of data at a speed of 6.1GB/s, leads to a latency of approximately $2330ms$ per token. Therefore, our method represents a substantial improvement over the baseline.

For a 16-bit model on a GPU machine, the flash load time is reduced to $40.5ms$, and memory management takes $40ms$, slightly higher due to the additional overhead of transferring data from CPU to GPU. Nevertheless, the baseline method's I/O time remains above 2000 milliseconds.

Detailed comparisons of how each method impacts performance are provided in Table 2.

4.2 Results for Falcon 7B Model

To verify that our findings generalize beyond OPT models we also apply the idea of LLM in flash to Falcon model. Since, the base line Falcon model is not sparse, we used a sparsified (relufied) version with almost the same performance as that of the base version (Mirzadeh et al., 2023). Similar to previous section, we present the results obtained

Table 2: The I/O latency of OPT 6.7B 16 bit on M1 max for different techniques when half the memory is available

Configuration				Performance Metrics			
Hybrid	**Predictor**	**Windowing**	**Bundling**	**DRAM (GB)**	**Flash→ DRAM(GB)**	**Throughput (GB/s)**	**I/O Latency (ms)**
✗	✗	✗	✗	0	13.4 GB	6.10 GB/s	2130 ms
✓	✗	✗	✗	6.7	6.7 GB	6.10 GB/s	1090 ms
✓	✓	✗	✗	4.8	0.9 GB	1.25 GB/s	738 ms
✓	✓	✓	✗	6.5	0.2 GB	1.25 GB/s	164 ms
✓	✓	✓	✓	6.5	0.2 GB	2.25 GB/s	87 ms

under the condition that approximately half of the model size is available for use in DRAM.

Predictors. In the Falcon 7B model, predictors of rank $r = 256$ are used for the initial 28 layers, and $r = 1152$ for the last four layers.

Window Configuration. Our model reserves memory for a window containing the last 4 tokens. This setup utilizes 33% of the Feed Forward Network (FFN). In terms of memory allocation, embeddings take 4.2% of the model size, attention weights account for 19.4%, and predictors require 4%. The active portion of the FFN, given our window size, is 25.3% (calculated as 0.33×76.8). Overall, this amounts to 52.93% of the model's total size.

Latency Analysis. Using a window size of 4 in our model requires accessing 3.1% of the Feed Forward Network (FFN) neurons for each token. In a 32-bit model, this equates to a data chunk size of 35.5 KiB per read (calculated as $2d_{\text{model}} \times 4$ bytes). On an M1 Max device, the time taken to load this data from flash memory is approximately 161ms, and the memory management process adds another 90ms, leading to a total latency of 250ms per token. In comparison, the baseline latency is around 2330 milliseconds, making our method approximately 9 to 10 times faster.

5 Related Works

Efficient Inference for Large Language Models. As LLMs grow in size, reducing their computational and memory requirements for inference has become an active area of research. Approaches broadly fall into two categories: model compression techniques like pruning and quantization (Han et al., 2016b; Sun et al., 2023; Jaiswal et al., 2023; Xia et al., 2023), (Zhang et al., 2022a; Xu et al., 2023; Shao et al., 2023; Lin et al., 2023; Hoang et al., 2023; Zhao et al., 2023; Ahmadian et al., 2023; Liu et al., 2023a; Li et al., 2023), and selective execution like sparse activations (Liu et al., 2023b), (Mirzadeh et al., 2023) or conditional computation (Graves, 2016; Baykal et al., 2023). Our work is complementary, focusing on minimizing data transfer from flash memory during inference.

Selective Weight Loading. Most related to our approach is prior work on selective weight loading. SparseGPU (Narang et al., 2021) exploits activation sparsity to load a subset of weights for each layer. However, it still requires loading from RAM. Flexgen (Sheng et al., 2023) offloads the weights and kv-cache from GPU memory to DRAM and DRAM to flash memory, in contrast we consider only the cases the full model can't reside in the whole DRAM and GPU memory on the edge devices. Flexgen is theoretically bound by the slow throughput of flash to DRAM in such scenarios. Firefly (Narang et al., 2022) shares our goal of direct flash access but relies on a hand-designed schedule for loading. In contrast, we propose a cost model to optimize weight loading. Similar techniques have been explored for CNNs (Parashar et al., 2017), (Rhu et al., 2013). Concurrently, Adapt (Subramani et al., 2022) has proposed adaptive weight loading for vision transformers. We focus on transformer-based LLMs and introduce techniques like neuron bundling tailored to LLMs.

To hide flash latency, we build on speculative execution techniques like SpAtten (Dai et al., 2021), (Bae et al., 2023). However, we introduce lightweight speculation tailored to adaptive weight loading.

Hardware Optimizations. There is a rich body of work on hardware optimizations for efficient LLM inference, including efficient memory architectures (Agrawal et al., 2022), (Gao et al., 2022), dataflow optimizations (Han et al., 2016a), (Shao et al., 2022), hardware evaluation frameworks Zhang2023AHE, and flash optimizations (Ham et al., 2016), (Meswani et al., 2015). We focus on algorithmic improvements, but these could provide additional speedups.

Speculative Execution. Speculative decoding (Leviathan et al., 2022; Zhang et al., 2023; He et al.,

2023) is a technique that uses a draft model for generation and uses the larger model to verify those tokens. This technique is orthogonal to us and can be used for further improvement. In case of speculative decoding, the window in our method should be updated with multiple tokens rather one.

Mixture of Experts. Mixture of Experts (Yi et al., 2023) have a sparse structure in their feed forward layer. This property can be used to combine with our method for enabling larger MoEs on device.

In summary, we propose algorithmic techniques to minimize weight loading from flash memory during LLM inference. By combining cost modeling, sparsity prediction, and hardware awareness, we demonstrate 4-5x and 20-25x speedup on CPU and GPU, respectively.

6 Conclusion

In this study, we have tackled the significant challenge of running large language models (LLMs) on devices with constrained memory capacities. Our approach, deeply rooted in the understanding of flash memory and DRAM characteristics, represents a novel convergence of hardware-aware strategies and machine learning. By developing an inference cost model that aligns with these hardware constraints, we have introduced two innovative techniques: 'windowing' and 'row-column bundling.' These methods collectively contribute to a significant reduction in the data load and an increase in the efficiency of memory usage.

The practical outcomes of our research are noteworthy. We have demonstrated the ability to run LLMs up to twice the size of available DRAM, achieving an acceleration in inference speed by 4-5x compared to traditional loading methods in CPU, and 20-25x in GPU. This breakthrough is particularly crucial for deploying advanced LLMs in resource-limited environments, thereby expanding their applicability and accessibility.

Our work not only provides a solution to a current computational bottleneck but also sets a precedent for future research. It underscores the importance of considering hardware characteristics in the development of inference-optimized algorithms, suggesting a promising direction for further explorations in this domain. We believe as LLMs continue to grow in size and complexity, approaches like this work will be essential for harnessing their full potential in a wide range of devices and applications.

Acknowledgements

We'd like to thank Itay Sagron, Lailin Chen, Mahyar Najibi, Qichen Fu, Moin Nabi, Sachin Mehta, Mohammad Samragh, Matt Johnson, Etai Zaltsman, Lin Chang, Dominic Giampaolo, Taal Uliel, Aftab Munshi for the valuable discussion.

References

Udit Agrawal, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and William J Dally. 2022. Atomlayer: minimizing dram data movement for ultra-sparse models on gpus. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–238.

Arash Ahmadian, Saurabh Dash, Hongyu Chen, Bharat Venkitesh, Stephen Gou, Phil Blunsom, A. Ustun, and Sara Hooker. 2023. Intriguing properties of quantization at scale. *ArXiv*, abs/2305.19268.

Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Maitha Alhammadi, Mazzotta Daniele, Daniel Heslow, Julien Launay, Quentin Malartic, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. 2023. The falcon series of language models: Towards open frontier models.

Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, Elton Zheng, Olatunji Ruwase, Shaden Smith, Minjia Zhang, Jeff Rasley, et al. 2022. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE.

Sangmin Bae, Jongwoo Ko, Hwanjun Song, and Se-Young Yun. 2023. Fast and robust early-exiting framework for autoregressive language models with synchronized parallel decoding. *ArXiv*, abs/2310.05424.

Cenk Baykal, Dylan Cutler, Nishanth Dikkala, Nikhil Ghosh, Rina Panigrahy, and Xin Wang. 2023. Alternating updates for efficient transformers. *ArXiv*, abs/2301.13310.

Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2023. Petals: Collaborative inference and fine-tuning of large models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 558–568, Toronto, Canada. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind

Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Han Dai, Yi Zhang, Ziyu Gong, Nanqing Yang, Wei Dai, Eric Song, and Qiankun Xie. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *Advances in Neural Information Processing Systems*, volume 34.

Mingyu Gao, Jie Yu, Wentai Li, Michael C Dai, Nam Sung Kim, and Krste Asanovic. 2022. computedram: In-memory compute using off-the-shelf dram. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1065–1079.

Alex Graves. 2016. Adaptive computation time for recurrent neural networks. In *International Conference on Machine Learning*, pages 3500–3509. PMLR.

Jongmin Ham, Jinha Kim, Jinwoong Choi, Cheolwoo Cho, Seulki Hong, Kyeongsu Han, and Taejoo Chung. 2016. Graphssd: a high performance flash-based storage system for large-scale graph processing. In *2016 USENIX Annual Technical Conference (USENIXATC 16)*, pages 243–256.

Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016a. Eie: efficient inference engine on compressed deep neural network. *arXiv preprint arXiv:1602.01528*.

Song Han, Huizi Mao, and William J Dally. 2016b. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*.

Zhenyu He, Zexuan Zhong, Tianle Cai, Jason D Lee, and Di He. 2023. Rest: Retrieval-based speculative decoding. *ArXiv*, abs/2311.08252.

Duc Nien Hoang, Minsik Cho, Thomas Merth, Mohammad Rastegari, and Zhangyang Wang. 2023. (dynamic) prompting might be all you need to repair compressed llms. *ArXiv*, abs/2310.00867.

Ajay Jaiswal, Zhe Gan, Xianzhi Du, Bowen Zhang, Zhangyang Wang, and Yinfei Yang. 2023. Compressing llms: The truth is rarely pure and never simple. *ArXiv*, abs/2310.01382.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2022. Fast inference from transformers via speculative decoding.

Jiaxi Li and Wei Lu. 2023. Contextual distortion reveals constituency: Masked language models are implicit parsers. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5208–5222, Toronto, Canada. Association for Computational Linguistics.

Liang Li, Qingyuan Li, Bo Zhang, and Xiangxiang Chu. 2023. Norm tweaking: High-performance low-bit quantization of large language models. *ArXiv*, abs/2309.02784.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. Awq: Activation-aware weight quantization for llm compression and acceleration. *ArXiv*, abs/2306.00978.

Zechun Liu, Barlas Oğuz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. 2023a. Llm-qat: Data-free quantization aware training for large language models. *ArXiv*, abs/2305.17888.

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

Moinuddin K Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel Loh. 2015. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 383–394. IEEE.

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.

Sharan Narang, Logan Feistel, Erich Elsen Undersander, Cindy Song, and Gregory Diamos. 2022. Firefly: A lightweight system for running multi-billion parameter models on commodity hardware. In *2022 ACM/IEEE 49th Annual International Symposium on Computer Architecture (ISCA)*, pages 757–771. IEEE.

Sharan Narang, Erich Elsen Undersander, and Gregory Diamos. 2021. Sparse gpu kernels for deep learning. In *International Conference on Learning Representations*.

Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. Timeloop: A systematic approach to dnn accelerator evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 241–251. IEEE.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *SC21: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14.

Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W Keckler. 2013. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page Article 13. IEEE Computer Society.

Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqiang Li, Kaipeng Zhang, Peng Gao, Yu Jiao Qiao, and Ping Luo. 2023. Omniquant: Omnidirectionally calibrated quantization for large language models. *ArXiv*, abs/2308.13137.

Yifan Shao, Mengjiao Li, Wenhao Cai, Qi Wang, Dhananjay Narayanan, and Parthasarathy Ranganathan. 2022. Hotpot: Warmed-up gigascale inference with tightly-coupled compute and reuse in flash. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 335–349.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single GPU. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 31094–31116. PMLR.

Vedant Subramani, Marios Savvides, Li Ping, and Sharan Narang. 2022. Adapt: Parameter adaptive token-wise inference for vision transformers. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. 2023. A simple and effective pruning approach for large language models. *ArXiv*, abs/2306.11695.

Haojun Xia, Zhen Zheng, Yuchao Li, Donglin Zhuang, Zhongzhu Zhou, Xiafei Qiu, Yong Li, Wei Lin, and Shuaiwen Leon Song. 2023. Flash-llm: Enabling low-cost and highly-efficient large generative model inference with unstructured sparsity. *Proc. VLDB Endow.*, 17:211–224.

Zhaozhuo Xu, Zirui Liu, Beidi Chen, Yuxin Tang, Jue Wang, Kaixiong Zhou, Xia Hu, and Anshumali Shrivastava. 2023. Compress, then prompt: Improving accuracy-efficiency trade-off of llm inference with transferable prompt. *ArXiv*, abs/2305.11186.

Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. 2023. Edgemoe: Fast on-device inference of moe-based large language models. *ArXiv*, abs/2308.14352.

Jinchao Zhang, Jue Wang, Huan Li, Lidan Shou, Ke Chen, Gang Chen, and Sharad Mehrotra. 2023. Draft & verify: Lossless large language model acceleration via self-speculative decoding. *ArXiv*, abs/2309.08168.

Shizhao Zhang, Han Dai, Tian Sheng, Jiawei Zhang, Xiaoyong Li, Qun Xu, Mengjia Dai, Yunsong Xiao, Chao Ma, Rui Tang, et al. 2022a. Llm quantization: Quantization-aware training for large language models. In *Advances in Neural Information Processing Systems*, volume 35.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022b. OPT: open pre-trained transformer language models. *CoRR*, abs/2205.01068.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. 2023. Atom: Low-bit quantization for efficient and accurate llm serving. *ArXiv*, abs/2310.19102.