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

Keivan Alizadeh, Iman Mirzadeh[*] , Dmitry Belenko[*] , S. 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 substantial 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 in flash memory, but bringing them on demand to DRAM. Our method involves constructing an inference cost model that takes into account the characteristics of flash memory, 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 hardware-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 up to 4x and 20x 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) have demonstrated strong performance across a wide range of natural language tasks (Brown et al., 2020; Chowdhery et al., 2022; Touvron et al., 2023a; Jiang et al., 2023; Gemini Team, 2023).

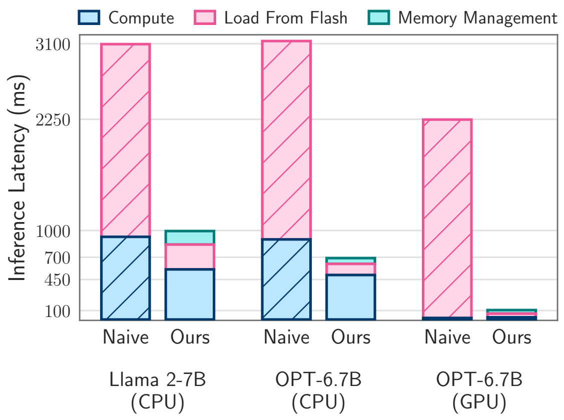


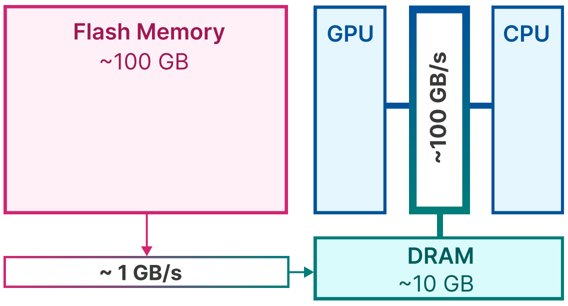
Figure 1: Average inference latency for a single token when only half of the model's memory is available: Our method selectively loads parameters on demand for each token generation step. The latency represents the time required to repeatedly load parameters from flash memory, combined with the time needed for computations.

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, which makes them challenging to load and run efficiently, especially on personal devices.

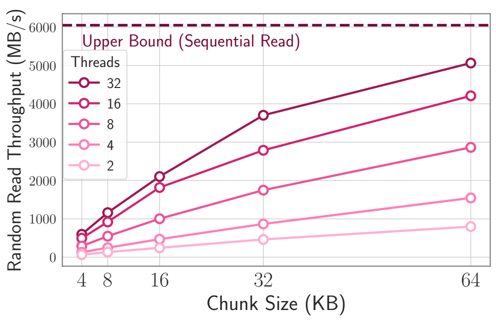
Currently, the standard approach is to load the entire model into DRAM (Dynamic Random Access Memory) for inference (Rajbhandari et al., 2021; Aminabadi et al., 2022). However, this severely limits the maximum model size that can be run. For example, a 7 billion parameter model requires over 14GB of memory just to load the parameters in half-precision floating point format, exceeding the capabilities of most personal devices such as smartphones. While it is possible to employ techniques such as quantization to reduce the model size, still, this cannot address the main limitation of loading the entire model into DRAM.

To address this limitation, we propose to store the model parameters in flash memory, which is

[*] Major Contribution
[†] {kalizadehvahid, imirzadeh, d_belenko, skhatamifard, minsik, cdelmundo, mrastegari, 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.

at least an order of magnitude larger than DRAM. Then, during inference, we directly load the required subset of parameters from the flash memory, avoiding the need to fit the entire model in DRAM. To this end, our work makes several contributions:

- First, we study the hardware characteristics of storage systems (e.g., flash, DRAM). We show that hardware constraints such as capacity and bandwidth limitations can have significant considerations when designing efficient algorithms for serving LLMs from flash (Section 2).
- Motivated by our findings, we propose several techniques that can help with (i) reducing the required data transfer, (ii) increasing the transfer throughput, and (iii) managing loaded parameters efficiently in DRAM (Section 3).
- Finally, as partially demonstrated in Figure 1, we show that our proposed techniques for optimizing the cost model and selectively loading parameters on demand allows us to run models 2x larger than the device's DRAM capacity and speed up inference up to 4x, 7x, and 20x compared to naive implementation in CPU, Metal and NVIDIA GPU backends, respectively (Section 4).

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. We aim to understand 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 modern NAND flash memories offer high bandwidth and low latency, they fall well short of the performance levels of DRAM (Dynamic Random-Access Memory), in terms of both latency and throughput. 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.

Load times for the models can be a problem even in the traditional DRAM-resident setup where weights are not reloaded partially – the initial, full load 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 the response latency.

2.2 Read Throughput

Flash memory systems perform optimally with large sequential reads. For instance, benchmarks on an Apple MacBook M1 Max with 1TB flash memory demonstrate speeds exceeding 6 GiB/s for a 1GiB linear read of an uncached file. However, this high bandwidth cannot be achieved 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. For smaller blocks, a substantial part of the overall read time is spent waiting for data transfer to begin. This is often referred to as latency to first byte. This latency reduces the overall throughput of each read operation considerably because the overall measured throughput has to take into account not just the speed of transfer once it begins, but the latency before it begins as well, which penalizes small reads. This means that if we coalesce the reads for rows and columns of the FFN matrices, we can pay the latency cost only once for any given row/column pair in both matrices and higher throughput can be realized. This principle is depicted in Figure 2b. Perhaps a counterintuitive yet interesting observation is that in some scenarios, it will be worthwhile to read more than needed (but in larger chunks) and then discard, rather than only reading strictly the 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 modern hardware using 32KiB or larger random reads across multiple threads.

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 DRAM 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 areas:

1. **Reducing Data Load**: Aiming to decrease latency associated with flash I/O operations by loading less data[1].
2. **Optimizing Data Chunk Size**: Enhancing flash throughput by increasing the size of data chunks loaded, thereby mitigating latency.
3. **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 optimizing the compute, as it is orthogonal to the core concerns of our work. Instead, we concentrate on optimizing flash memory interactions and memory management to achieve efficient inference on memory-constrained devices. We will elaborate on the implementation details of these strategies in the experimental setup section.

3.1 Reducing Data Transfer

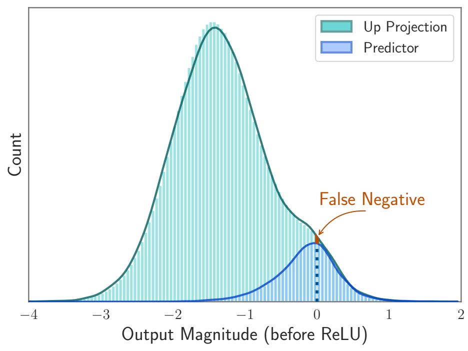
Our method leverages the inherent activation sparsity found in Feed-Forward Network (FFN) models, as documented in preceding research. The OPT 6.7B model, for instance, exhibits a notable 97% sparsity within its FFN layer. Similarly, the Falcon 7B model has been adapted through fine-tuning, which involves swapping their activation functions to ReLU, resulting in 95% sparsity while being similar in accuracy (Mirzadeh et al., 2023). Replacing activations of Llama 2 model (Touvron et al., 2023b) by FATReLU and finetuning can achieve 90% sparsity(Song et al., 2024). In light of this information, our approach involves the iterative transfer of only the essential, dynamic subset of the weights from flash memory to DRAM for processing during inference.

Selective Persistence Strategy. We opt to retain the embeddings and matrices within the attention mechanism of the transformer constantly in DRAM. For the Feed-Forward Network (FFN) portions, only the non-sparse segments are dynamically loaded into DRAM as needed. Keeping 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.

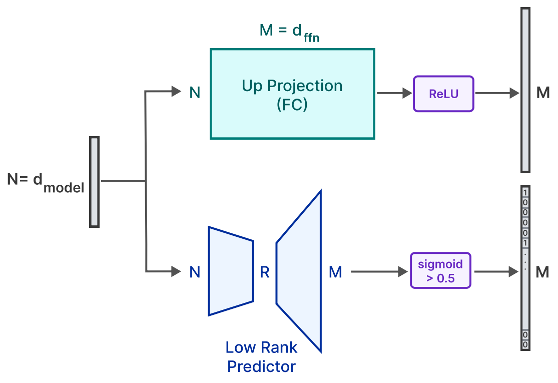
Anticipating ReLU Sparsity. The ReLU activation function naturally induces over 90% sparsity in the FFN's intermediate outputs, which reduces the memory footprint for subsequent layers that utilize these sparse outputs. However, the preceding layer, namely the up project, must be fully present in memory.

To avoid loading the entire up projection matrix, we follow Liu et al. (2023b), and employ a

[1]It is notable that, by *data* we often refer to the weights of the neural network. However, the techniques we have developed 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 the preactivation of elements that the predictor detected as 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.

Table 1: The low-rank predictor has a marginal impact on zero-shot metrics as the 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

low-rank predictor to identify the elements zeroed by ReLU (see Figure 3b). We used a balanced loss over negative and positive samples of each layer. 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 the current layer is sufficient for hardware-aware weight-loading algorithm design but leads to more accurate outcomes due to deferred inputs. We used 10000 samples from the C4 training dataset to do the training for 2 epochs. It took 4 hours on an A100 GPU to train each predictor.

We thereby only load elements indicated by the predictor, as shown in Figure 3a. Furthermore, as demonstrated in Table 1, using predictors does not adversely affect the model's performance in 0-shot tasks. For more details please refer to Appendix B.

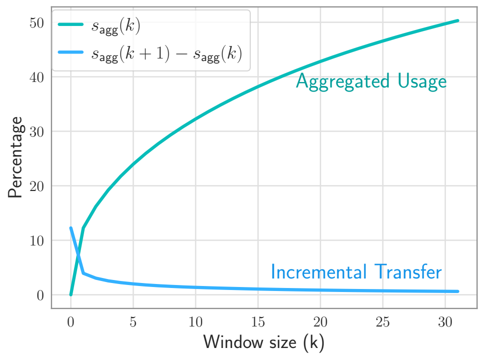
The Sliding Window Technique. In our study, we define an *active neuron* as one that yields a positive output in our low-rank predictor model. Our approach focuses on managing neuron data by employing a *Sliding Window Technique*. This technique entails maintaining a DRAM cache of only the weight rows that were predicted to be required by the recent subset of input tokens. The key aspect of this technique is the incremental 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 cached weights required by tokens that are no longer within the sliding window (as depicted in Figure 4b).

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 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), while smaller values of k can help conserve DRAM that is used to store the cached weights. In determining the size of the sliding window, the aim is to maximize it within the constraints imposed by the available memory capacity.

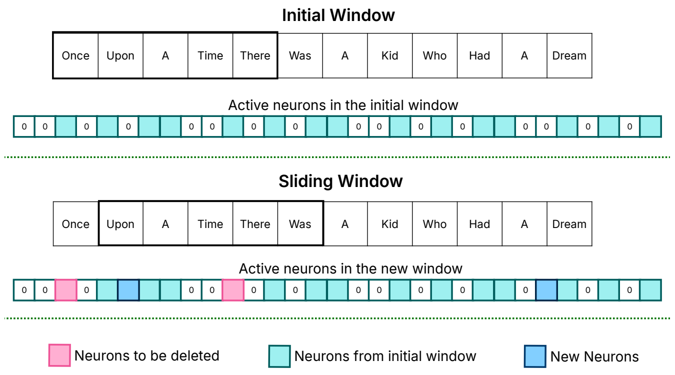
3.2 Increasing Transfer Throughput

To increase data throughput from flash memory, it is crucial to read data in larger chunks, preferably sized as the multiples of the block size of the underlying storage pool. 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. Note that in the FFN layer, the usage of the ith column from the up projection and the ith row from the down projection



(a) aggregated neuron usage



Initial Window
Once Upon A Time There Was A Kid Who Had A Dream
Active neurons in the initial window
Sliding Window
Active neurons in the new window
Neurons to be deleted
Neurons from initial window
New Neurons

(b) sliding window

Figure 4: **(a)** Aggregated neuron usage of the tenth layer of Falcon 7B: the slope of aggregated neuron usage is decreasing. Other layers exhibit the same pattern. **(b)** Rather than deleting neurons that were brought to DRAM we keep the active neurons of past k tokens (we use $k = 5$): when the new token "Was" is being processed only a small fraction of new weights need to be loaded.

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 5 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 5. Our analysis and experiment show this increases the throughput of the model.

Bundling Based on Co-activation. We hypothesized that neurons might exhibit highly correlated activity patterns, enabling bundling. By analyzing activations on the C4 validation dataset, we found a power law distribution of coactivations. However, bundling neurons with their highest coactivated neuron (closest friend) led to multiple loadings of highly active neurons, counteracting our goal. This result suggests that very active neurons are the closest friends of many others. We present this negative result to inspire future research on effective neuron bundling for efficient inference. Please refer to Appendix D for details.

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 neuron data in DRAM. This is particularly costly when a substantial portion (approximately 25%)

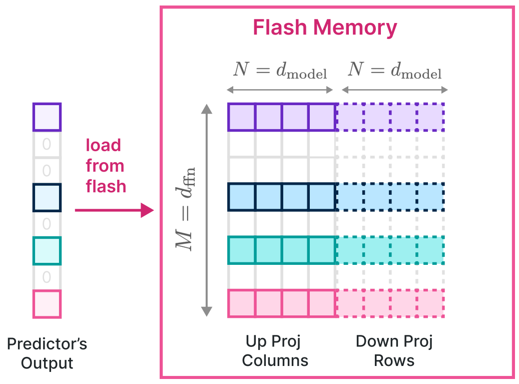


Figure 5: By bundling columns of the up project and rows of the down project layer, we can load 2x chunks instead of reading columns or rows separately.

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 concatenated 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

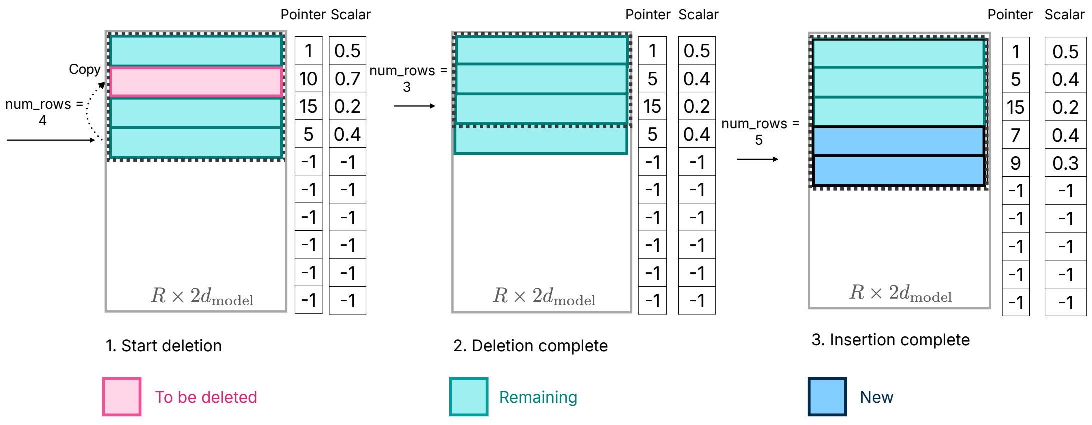


Figure 6: Memory management; First we replace elements to be deleted by last elements to maintain a consecutive occupation of memory. Then the new weights are stacked at the end. This reduces the unnecessary data movements.

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 enough memory for each layer in advance, we minimize the need for 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 can be done as depicted in Figure 6:

1. **Deleting Neurons:** Neurons that are no longer required are identified efficiently in linear time, utilizing the associated `last_k_active` value 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.
2. **Bringing in New Neurons:** The required weights are 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, reducing 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 FFN 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 Experiments and Results

We start this section by briefly discussing our experimental setup and implementation details. Next, we show that the techniques introduced in Section 3 can improve the inference latency significantly across different models and runtime platforms. We postpone the some details to the appendix sections as follows: performance of our trained low-rank predictor (Appendix B).

4.1 Experimental Setup

Our work is mainly motivated by optimizing inference efficiency on personal devices. To this end, in our experiments, 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. For the implementation of our inference process, we utilize HuggingFace Transformers library (Wolf et al., 2019) and PyTorch (Paszke et al., 2019). This setup is tested under the condition that approximately half of the model size is available in DRAM. While with a different level of sparsity or employing quantization, one can work with smaller available DRAM capacity, these optimizations are orthogonal to our proposed method.

Models. We mainly use OPT 6.7B (Zhang et al., 2022b) and the sparsified Falcon 7B (Mirzadeh et al., 2023) model for our evaluations, but we additionally report results on Phi-2 (Gunasekar et al., 2023), Persimmon 8B (Elsen et al., 2023) and a

Table 2: The I/O latency of OPT 6.7B 16 bit on M1 Max when half the memory is available. By employing the activation predictor and windowing, we can reduce the data transfer from flash memory to DRAM. While this reduces the throughput, the bundling technique can alleviate this by doubling the data transfer chunk size and hence the throughput which leads to reducing the overall latency to half.

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	2196 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

Llama 2 (Touvron et al., 2023b) which is sparsified using FATReLU (Song et al., 2024). Note that the techniques introduced in this work are mostly independent of architecture.

Data. We use a small subset of C4 validation dataset for our latency measurements. We take the first 128 tokens of each example as the prompt, and generate 256 new tokens.

Hardware Configuration. Our models are evaluated across three hardware setups. The first includes an Apple M1 Max with a 1TB SSD. The second features an Apple M2 Ultra with a 2TB SSD. On MacBooks we run the model on the CPU with float32 or GPU with Metal and float16. The third setup uses a Linux machine with a 24GB NVIDIA RTX 4090, where GPU computations utilize bfloat16 models. Across all setups, we assume nearly half of the total memory (DRAM and GPU) is allocated for model computations.

Baselines. We compare our models with a *naive* baseline of loading the model on demand when doing the forward pass. We additionally compare with our *hybrid* loading approach as a secondary baseline when half of the model is persisted in memory and the other half is loaded on demand at generation of every token without use of sparsity. We used best theoretical possible numbers for IO latency for each of the methods to make a fair comparison, the real number might be higher. 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. Given the nature of our setup (i.e., the limited available DRAM or GPU memory), we are not aware of any other method that can surpass this theoretical I/O efficiency.

Implementation Details. To optimize data loading from flash memory, our system employs reads parallelized over 32 threads. This multithreaded approach is intended to both better amortize latency to the first byte by not waiting for each read sequentially, and maximize read throughput by reading multiple streams at once (Figure 2b). To better assess the actual throughput, we conducted benchmarks without the aid of operating system caching leading to a more accurate measurement.

4.2 Faster Load From Flash

Our first result in Table 2 demonstrates the effectiveness of techniques we introduced in Section 3, where the I/O latency depends on how much data is being transferred from flash to DRAM, and the chunk size which determines the throughput. For instance, by using a low-rank predictor, we reduce the data transfer significantly, and the amount of this traffic can be further reduced using our proposed windowing technique. Compared to a long, contiguous read, scattered reads will necessarily result in lower throughput (e.g. 1.25 GiB/s sparse vs 6.1 GiB/s dense), but this is partially mitigated by bundling up-projection and down-projection weights. The overall effect of sparse reads is still strongly favorable, because only a small subset of the overall weights is loaded incrementally in each iteration, and the load of just the required subset of weights takes less time and less DRAM.

Additionally, we examine end-to-end latencies under various setups in Table 3. We allocate approximately 50 % of the model size for OPT, Falcon, and Persimmon and Llama 2. For the significantly smaller Phi-2 model, we observed less

Table 3: The end-to-end inference latency across different setups. Our efficient implementation (referred as *All*) that employs the predictor, windowing, and bundling can lead to significant latency reduction.

			Inference Latency (ms)			
Model	**Method**	**Backend**	**I/O**	**Mem**	**Compute**	**Total**
OPT 6.7B	Naive	CPU	2196	0	986	3182
OPT 6.7B	All	CPU	**105**	58	506	**669**
OPT 6.7B	Naive	Metal M1	2196	0	193	2389
OPT 6.7B	All	Metal M1	**92**	35	438	**565**
OPT 6.7B	Naive	Metal M2	2145	0	125	2270
OPT 6.7B	All	Metal M2	**26**	8	271	**305**
OPT 6.7B	Naive	GPU	2196	0	22	2218
OPT 6.7B	All	GPU	**30**	34	20	84
OPT 6.7B	Speculative	GPU	38.5	9.5	12	**60**
Falcon 7B	Naive	CPU	2295	0	800	3095
Falcon 7B	Hybrid	CPU	1147	0	800	1947
Falcon 7B	All	CPU	**161**	92	453	**706**
Persimmon 8B	Naive	CPU	2622	0	1184	3806
Persimmon 8B	Hybrid	CPU	1311	0	1184	2495
Persimmon 8B	All	CPU	**283**	98	660	**1041**
Phi-2 2.7B	Naive	CPU	885	0	402	1287
Phi-2 2.7B	Hybrid	CPU	309	0	402	711
Phi-2 2.7B	All	CPU	**211**	76	259	**546**
Llama 2 7B	Naive	CPU	2166	0	929	3095
Llama 2 7B	Hybrid	CPU	974	0	929	1903
Llama 2 7B	All	CPU	**279**	152	563	**994**

sparsity rates, prompting us to set this limit at 65%. We observe a significant improvement in loading efficiency over both naive and hybrid approaches across all models. Moreover we showed the GPU backend outcomes further improve when combined with speculative decoding.

4.3 The Memory-Latency Tradeoff

So far, we have mainly worked under the assumption that the available DRAM is roughly half of our model size. However, we note that this is not a hard constraint and we can relax this constraint.

To this end, we study the impact of window size on memory usage, and consequently on latency. By increasing the window size, we increase the percentage of model parameters that we keep in DRAM. As a result, we need to bring fewer parameters, and hence the latency can be reduced at the cost of using higher DRAM as shown in Figure 7.

5 Ablation analysis

5.1 The Impact of Longer Generation

In our previous results, we have used short to medium-length (256 tokens) generations for our benchmarks. It is possible that for longer generation of tokens, the ssd enable thermal throttling and lower the performance. However, Figure 8 shows that this is not the case, even when we generate

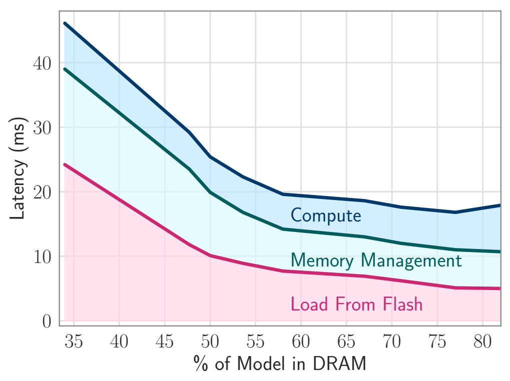


Figure 7: By bringing more of our model (OPT-6.7B) parameters into DRAM, the latency can be reduced on the GPU machine.

1000 tokens for OPT 6.7B model on GPU. Moreover, we show that the average flash latency doesn't increase as we go further in generation. In contrast, the flash latency for the the first few tokens is higher since the allocated memory in DRAM is empty and needs to be filled in with neurons and for first few tokens we need more data transfer.

Also it is possible to argue that the non-greedy sampling methods such as the Nucleus sampling (Holtzman et al., 2020) method can result in more diverse activation, and hence less favorable towards our method. We found out this is not the case either for long token generations. Nucleus sampling doesn't lead to lower performance in long generation in neither cpu or gpu.

5.2 Speculative Decoding

To further showcase the strength of our method and adaptability to other decoding strategies we have applied speculative decoding on the OPT 6.7B model. The challenge for doing speculative decoding is the limited memory available within DRAM. Given λ tokens from the draft model, the big model verifies them and will keep a window of size k for each layer. The model should decide neurons of which tokens to keep in memory before verifications are done. If the model keeps the last k tokens out of $\lambda + 1$ tokens in memory and most of them get rejected, there will be very few neuron reuse for the next forward pass. We conjecture that if the ratio of the acceptance is α keeping the last k tokens ending with $\alpha(\lambda + 1)$th token is optimal in DRAM. We used $\lambda = 4$ and were able to improve the speed of decoding by 1.4x as shown in table 5, this is close to the original 1.58x speedup of speculative decoding.

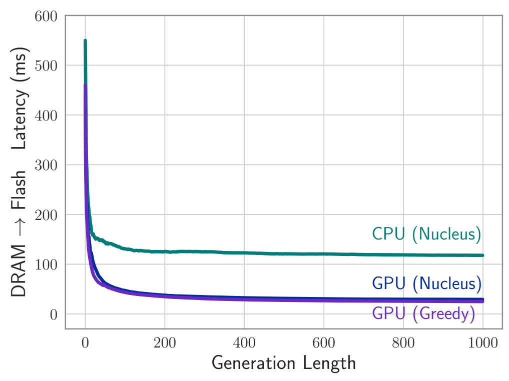


Figure 8: Weight loading latency of OPT 6.7B with increasing generation length.

5.3 A Note on Power Consumption

In evaluating the efficiency of our method, we compared the power consumption of our sparse model approach with that of generating tokens using a dense model of similar size. While the power usage (energy per unit of time) of the sparse model was lower than that of the dense model, the extended duration required for token generation resulted in the sparse model having a higher total energy consumption. This is going to be reflected in the greater area under the curve when plotting power over time for the sparse model compared to the dense model. A systematic and quantitative evaluation of the exact power usage pattern is left as a future work.

6 Related Works

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 such as 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; Li et al., 2023), and selective execution such as sparse activation (Liu et al., 2023b; Mirzadeh et al., 2023), or conditional computation (Graves, 2016; Baykal et al., 2023). Our work is orthogonal to these directions, focusing mainly on minimizing data transfer from flash memory during inference.

Perhaps most related to our work is the literature 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 where the full model can't reside in the whole DRAM and GPU memory on the edge devices. Notably, FlexGen is still theoretically bound by the slow throughput of flash to DRAM in such scenarios. An expanded discusion of related works is deferred to Appendix E.

Overall, the primary assumption in the literature is that the model can fully reside in the GPU memory or system DRAM. However, considering the limited resources available on personal devices, we do not share this assumption in this work. Instead, we concentrate on exploring how to store and load parameters on flash memory more efficiently, aiming to enhance inference efficiency.

7 Discussion

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 new techniques: 'windowing' and 'row-column bundling'.

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

While in this work we have studied the previously unexplored problem of serving LLMs from flash, we note that this work is only a first step in this direction, and has several limitations that we discuss in the next section, and we believe there are several interesting problems left to be explored in future works. For instance, from the algorithmic perspective, more optimized techniques of weight bundling and data structures can be crafted, while from the engineering perspective, the characteristics of specific hardware platforms can inform works on building more efficient inference stacks.

8 Limitations

Our study represents an initial endeavor in the pursuit of democratizing Large Language Model (LLM) inference, making it accessible to a wider array of individuals and devices. We recognize that this early effort has its limitations, which, in turn, open up compelling avenues for future research. A critical aspect for future exploration is the systematic analysis of power consumption and thermal limitations inherent in the methods we propose, particularly for on-device deployment.

Currently, our study is limited to single-batch inference. We provide some preliminary results on combining our proposed idea with speculative decoding, however, expanding this to include more complex scenarios like prompt processing and multi-batch inference are valuable areas for further investigation.

In our initial proof of concept, we operated under the assumption of memory availability being half the size of the model. Exploring the dynamics of working with varying memory sizes—both larger and smaller—introduces a fascinating balance between latency and accuracy, and is a compelling area for future exploration.

In conclusion, our methodology is constructed on the foundation of sparsified networks. Nonetheless, the underlying concept holds potential for broader applications. It can be adapted to selectively load weights in non-sparse networks or to dynamically retrieve model weights from flash storage. This adaptation would be contingent on the specific requirements of the input prompt or the contextual parameters provided. Such an approach suggests a versatile strategy for managing model weights, and optimizing performance based on the nature of the input, thereby enhancing the efficiency, usefulness, and applicability of the proposed scheme in various scenarios dealing with Large Language Models (LLMs).

Acknowledgements

We would like to thank Itay Sagron, Lailin Chen, Chenfan (Frank) Sun, Hanieh Hashemi, Mahyar Najibi, Qichen Fu, Moin Nabi, Peter Zatloukal, Arsalan Farooq, Sachin Mehta, Mohammad Samragh, Matt Johnson, Etai Zaltsman, Lin Chang, Dominic Giampaolo, Tal Uliel, Hadi Pouransari, Fartash Faghri, Oncel Tuzel, Samy Bengio, Ruoming Pang, Chong Wang, Ronan Collobert, David Grangier, and Aftab Munshi for the valuable discussions.

References

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.

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.

Yew Ken Chia, Pengfei Hong, Lidong Bing, and Soujanya Poria. 2023. Instructeval: Towards holistic evaluation of instruction-tuned large language models.

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.

Erich Elsen, Augustus Odena, Maxwell Nye, Sağnak Taşırlar, Tri Dao, Curtis Hawthorne, Deepak Moparthi, and Arushi Somani. 2023. Releasing Persimmon-8B.

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.

Google Gemini Team. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

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

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. 2023. Textbooks are all you need. *CoRR*, abs/2306.11644.

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)*.

Awni Hannun, Jagrit Digani, Angelos Katharopoulos, and Ronan Collobert. 2023. MLX: Efficient and flexible machine learning on apple silicon.

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

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

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.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

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.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *CoRR*, abs/2310.06825.

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

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 Oguz, 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. *CoRR*.

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.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035.

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.

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.

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.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. 2019. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771.

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.

Zhengyan Zhang, Yixin Song, Guanghui Yu, Xu Han, Yankai Lin, Chaojun Xiao, Chenyang Song, Zhiyuan Liu, Zeyu Mi, and Maosong Sun. 2024. Relu^2 wins: Discovering efficient activation functions for sparse llms.

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.

A Appendix Overview

The appendix is structured as follows:

- In Appendix B, we provide additional details on the low-rank predictor introduced in Section 3. We evaluate our trained predictors from both accuracy (i.e., their impact on the model's accuracy) and efficiency perspectives (i.e., the additional neurons they predict to be activated).
- Appendix C offers a more detailed description of our experimental setup and implementation for the experiments conducted in Section 4.
- In Appendix D, we discuss a negative result regarding the strategy of bundling neurons based on co-activation as a potential method for increasing chunk size (cf. Section 3.2). We intentionally include this negative result as we believe it may inspire future research on effective neuron bundling and its utilization for efficient inference.
- In Appendix E, we delve deeper into the review of related works in the literature.
- In Appendix F, we go over implications of llm in flash when going to smaller devices.
- Finally, Appendix G compares the texts generated by the base model with those produced by our models that utilize the predictor.

B Low-Rank Activation Predictor: Additional Results

B.1 Sparsity patterns of predictors

The number of neurons predicted to be active will determine the efficiency of our algorithm, the less sparse the predicted activation the more weights will have to be loaded from flash. We evaluated the sparsity patterns over 100 random samples of the C4 validation dataset. In Figure 9 we can see the sparsity patterns of OPT, Persimmon, and Phi. In OPT, the number of active neurons predicted by the predictor is 3x the amount of actual sparsity observed in the case of dense inference. In Persimmon it is about the same - 3x the required neurons, and in Phi-2 it is roughly 2x the required neurons of the original model that are activated by the predictor. The neurons that are activated by the model and not the predictor are the false negatives. The gap between the neurons active in both the predictor and the model, and the neurons active only in the model is very narrow in all three models, hence false negatives constitute a small fraction of predictions. To reduce false negatives, the predictor has to "over-predict", which results in loading neurons that are redundant, that is, will have zero activation and no effect on the outcome. An interesting future direction of this work is improving the accuracy of the predictors to be able to load fewer neurons. One observation we had in OPT and Persimmon is the later layers have more active neurons, which can be seen in Figure 9d.

B.2 Accuracy of models using predictors

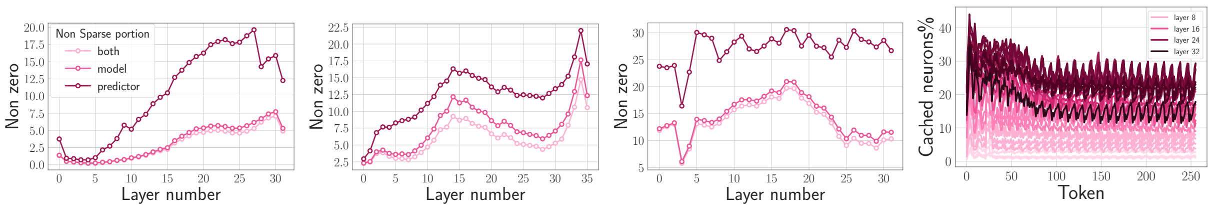
We evaluate the accuracy of models on public benchmarks with predictors in place. In Table 4 it can be seen zero shot accuracy of models doesn't drop. Also, we can see that increasing the predictor size for the last 4 layers of Persimmon and Falcon improves the zero-shot metrics. We evaluated models on MMLU (Hendrycks et al., 2021) benchmark as well. We used Instruct Eval's implementaion (Chia et al., 2023) for evaluating MMLU. In Figure 10a we can see the MMLU of Persimmon doesn't drop when the last 4 layers use higher rank predictors but this is not the case for lower ranked ones. Phi2's MMLU will drop 2.3 points from the relufied model still keeping at 52 as shown in Figure 10b. By increasing the threshold of low-rank predictor we can reduce the amount of data load, this comes with a slight degradation in zero-shot metrics as seen in the Table 4 for different thresholds of the Persimmon model. We have used threshold=0.7 for Persimmon.

B.3 Overhead of predictors

The average rank of predictors in the OPT-6.7B is 240, this will result in less than 2.4% of non-embedding weights and FLOPs. In M1 Max CPU experiments this was comprising 2.75% and in RTX GPU it was 4.8% of inference time which is negligible. For Falocn 7B, predictors take 4% model size and CPU computation. For Persimmon it was taking 2.85% of inference time on CPU. For Llama 2 7B it was taking 3.92% of inference time on CPU.

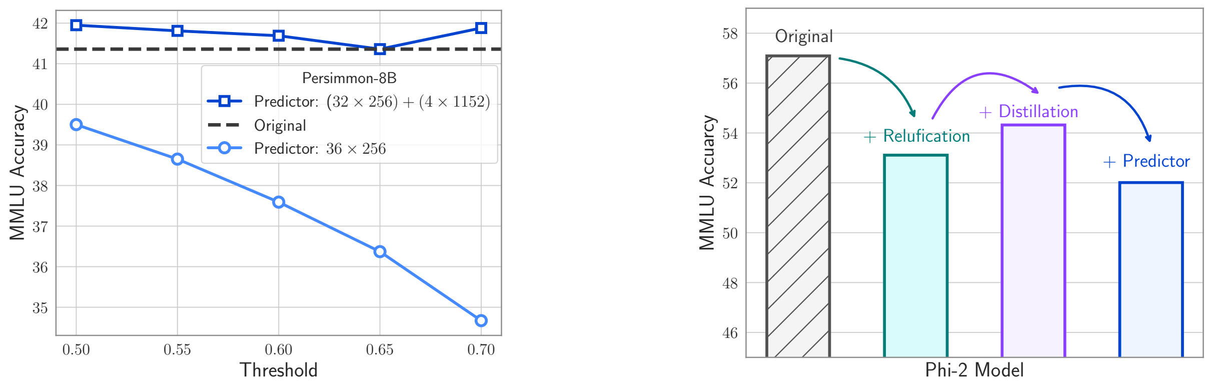
C Extended Results

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



(a) OPT-6.7B (b) Persimmon 8B (c) Phi-2 (d) Rows cached over time

Figure 9: **(a)** The percentage of fired neurons in each layer's FFN is less than 5%. In predictor, roughly 3x of this amount will get activated. The narrow gap between neurons that are activated in both shows the model output will not change abruptly. **(b)** Earlier layers of Persimmon have less active neurons, and later layers of Persimmon have higher active neurons, so we trained larger predictors for them. **(c)** In Phi2 middle layers have a higher active neuron ratio, so we trained larger predictors for those layers. **(d)** The number of neurons cached within a real scenario of inference, later layers have more cached rows because of their higher nonsparse ratio.



(a) (b)

Figure 10: **(a)** If we use larger predictors in the last 4 layers MMLU wouldn't drop a lot in Persimmon. **(b)** Phi's MMLU will drop in the relufication process due to lower quality data, using distillation can improve the results for that. Using predictors will downgrade the results but still keep it at 52.

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]

For the implementation of our inference process, we utilize the HuggingFace Transformers and KV caching. This setup is tested under the condition that 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, or alternatively use larger models. Such a configuration demonstrates the practicality of executing inference with lower memory footprints.

Systems performance optimization: The primary target of our experiments was the Apple macOS 14.3 operating system. For high-performance inference, most of the existing deep learning frameworks require that the shape of the weights and the intermediate results in the computation remain static throughout. In particular, the Metal Performance Shaders (MPS) backend for PyTorch demonstrates rather steep performance cliffs when any shape dynamism is present in the computational graph. In order to build a high-performance implementation, we chose to borrow custom, dynamism-friendly Metal kernels from Apple's open-source MLX deep learning framework (Hannun et al., 2023). In addition, we made use of the unified memory architecture available on Apple systems, which we exploit to maintain the weights cache using the GPU allocator, by creating the tensors using the `MTLStorageModeShared` allocation mode. This mode allows both the CPU and the GPU to access the same memory buffer directly, without redundant copies. We observe that the inputs to and the outputs from the feed-forward network

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

Table 4: Model performance on zero-shot tasks when using predictors

Model	**Predictor Parameters**			**Zero-Shot Metrics**		
	Rank for Sensitives [*]	**Rank for Other Layers**	**Threshold**	**ArcEasy**	**Arc Challenge**	**Hella Swag**
OPT 6.7B	-	-	-	66.1	30.6	50.3
OPT 6.7B with predictors	1024	128	0.5	66.2	30.6	49.8
Falcon 7B	-	-	-	74.62	40.05	57.77
Falcon 7B relufied	-	-	-	72.52	38.23	54.17
Falcon 7B relufied with predictors	128	128	0.50	70.20	35.41	50.74
Falcon 7B relufied with predictors	1152	128	0.50	71.51	34.22	52.28
Falcon 7B relufied with predictors	1152	256	0.50	72.35	36.35	53.16
Persimmon 8B	-	-	-	67.80	34.64	50.70
Persimmon 8B with predictors	256	256	0.5	67.26	33.87	50.51
Persimmon 8B with predictors	256	256	0.55	66.71	34.73	50.54
Persimmon 8B with predictors	256	256	0.60	66.67	34.04	50.59
Persimmon 8B with predictors	256	256	0.65	66.41	34.22	50.42
Persimmon 8B with predictors	1152	256	0.70	66.30	34.40	52.70
Phi-2	-	-	-	79.62	51.49	55.17
Phi-2 relufied	-	-	-	80.60	50.12	54.30
Phi-2 relufied with predictors	800	mix 160, 480	0.40	79.96	49.57	53.50
Phi-2 relufied with predictors	800	mix 160, 480	0.55	78.90	47.90	52.75

[*] For OPT, Falcon, and Persimmon sensitive layers are the last 4 layers. For Phi-2 it is the middle 8.

have a static shape, so by hiding the dynamism inside a binary extension and handling the shape dynamism and memory management internally, we were able to achieve a level of performance that is not achievable with PyTorch MPS backend alone while leaving the rest of the model intact. Over the course of our work, we were able to eliminate nearly all redundant data movement, improving inference performance.

Caching Considerations for Data Loading from Flash Memory. When data is read from flash memory, the operating system typically caches the blocks in the block cache, 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. Practical systems may or may not rely on filesystem cache, depending on requirements.

For the purpose of our hardware benchmarking in this study, we deliberately and significantly pessimize 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 lower bound of throughput in scenarios where no caching is permitted and makes the benchmarks repeatable. It's worth noting that these figures can improve if either the inference code or the operating system is allowed to cache some part of the weights.

While OS-level buffer caching is advantageous for general-purpose 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 a situation where the file system level cache does not help because in order to evaluate later layers earlier layers must be evicted in a rolling pattern, so the effective cache hit rate is close to zero. Aside from being inefficient, this can cause coexistence issues with other processes due to memory allocation pressure and Translation Lookaside Buffer (TLB) churn.

C.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

Table 5: The end-to-end inference latency across different setups with standard deviation. Our efficient implementation (referred to as *All*) that employs the predictor, windowing, and bundling can lead to significant latency reduction.

			Inference Latency (ms)			
Model	**Method**	**Backend**	**I/O**	**Mem**	**Compute**	**Total**
OPT 6.7B	All	CPU	**104.90 (± 18.46)**	57.79 (± 9.63)	506.50 (±17.33)	**669.20 (± 39.74)**
OPT 6.7B	All	GPU	**30.55 (±3.09)**	34.11 (±2.38)	19.97 (±0.86)	84.64 (±6.16)
OPT 6.7B	Speculative	GPU	38.53 (±10.0)	9.45 (±1.7)	12.18 (±2.0)	**60.16 (±13.4)**
Persimmon 8B	All	CPU	**310.52 (±41.12)**	155.80 (±21.30)	623.74 (± 24.76)	**1090.08 (±79.08)**
Phi-2	All	CPU	**211.08 (±24.81)**	76.87 (±7.18)	258.74 (±20.90)	**546.69 (±31.98)**

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 $k = 4$ 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 on 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 4, 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 the average latency of $105ms$ per token for loading from flash and $57ms$ for memory management (involving neuron deletion and addition). Thus, the total memory-related latency is less than $162ms$ 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 $2196ms$ 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 $30.5ms$, and memory management takes $35ms$, 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.

C.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 (Almazrouei et al., 2023). Since the original 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 the previous section, we present the results obtained 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 Net-

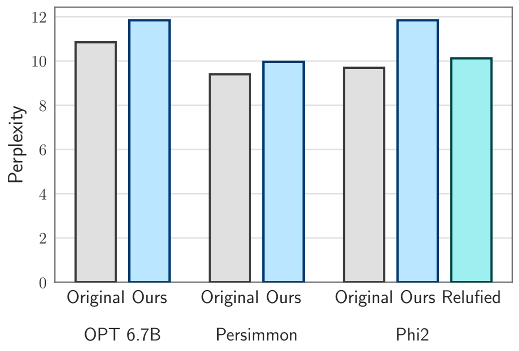


Figure 11: There is a slight drop in the perplexity of OPT and persimmon and more drop in phi-2 after using predictors.

work (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 2196 milliseconds, making our method approximately 9 to 10 times faster.

C.3 Persimmon 8B

We have applied LLM in Flash for Persimmon 8b models. Since Persimmon is already using squared ReLU activation we didn't need to finetune it further.

Predictors. In the Persimmon 8B base model, predictors of rank r=256 are used for the initial 32 layers and r = 1152 for the last four layers.s. Persimmon's sparsity is less than OPT and Falcon so we changed the sigmoid threshold to 0.7. In figure 10a you can see that the MMLU of the model doesn't drop with this setting. This wouldn't be the case if all the predictors had a rank of 256. Also in Figure 11 you can see that perplexity on wikitext2 with doesn't drop abruptly. Qualitative evaluations can be found in Section G.

Window configuration. Our model reserves memory for a window containing the last 4 tokens and also reduces window size dynamically whenever the whole memory usage passes the 25% of FFN size threshold.

Latency analysis. Since we have fixed the memory budget we won't exceed the 25% limit in the FFN which will be 50% of the total model size. We used nucleus sampling «cite nucleus» with p=0.9 to have a broader analysis. As it can be seen in Figure 1 it takes 310ms for loading from flash and 155ms for memory management.

C.4 Phi-2

We have applied LLM in Flash for Phi-2 models. We first relufied the model then trained the predictor and applied inference. Since the model is already small, we gave it 65% of its memory for running the inference. During the inference, we modified the window size to make sure it will never exceed the limit.

Relufication. We finetuned the model using a refined-web dataset following Mirzadeh et al. (2023). We found that adding a distillation loss as suggested by (Liu et al., 2023a) improves the results as can be seen in 10b. MMLU metric drops from 57 to 54.3 after relufication with distillation.

Predictors. The sparsity pattern of Phi-2 is different than other models. As you can see in figure 9c the sparsity of the middle layers is less than other layers for a random sample of the C4 dataset. As a general rule of thumb, we trained larger predictors for the less sparse layers. If layers are grouped by 4, we will have 8 groups of layers. For the last group, we didn't use any predictors. For the first, second, and seventh groups, we trained a predictor of size 160. For the third and sixth groups we trained predictors of size 480 and for groups in the middle we trained predictors of size 800.

Latency analysis. Phi-2 gets 2.35x speedup over naive baseline as it can be seen in table 3. It also improves our hybrid-only approach.

C.5 Llama 2

To further validate our result we tried running Llama2 (Touvron et al., 2023b) on flash. We used the sparisified Llama2 (Song et al., 2024) as the base model and run our experiments on M1 Max CPU. We used window size of 2. We didn't cache weights when the total memory was growing over 55% of model size.

Sparse models. The sparsified model (Song et al., 2024) uses FATReLU function to ensure sparsity of llama is above 90%. For models that have used Swi-GLU activation function (having a gated linear unit, a down project and an up project), replacing Swish with ReLU within the FFN doesn't ensure high amount of sparsity (Mirzadeh et al., 2023). The FATReLU function activates neurons with gated value greater than a threshold. This will ensure only a small portion of neurons are activated which are the most informative.

Predictors. We used predictors of size 1024 in 4 middle layers and predictor of size 256 in all other layers. The reason we used larger predictors in the middle layers is higher neuron activation in middle layers (similar to Phi2). The reason why in some networks middle layers are more active and in some networks later layers are more active is subject to follow up research.

Latency analysis. LLM in flash gets 3x speed up over naive baseline (Table 3). It is also performing better than hybrid model which is the theoretical lower bound for approaches that doesn't use sparsity.

Accuracy analysis. When doing MMLU evaluation using InstructEval repo (Chia et al., 2023) we got MMLU of 41.8 for Llama 2, 38.96 for sparsified model by (Song et al., 2024) and 38.63 after training our predictors. We noted a difference between reported numbers and our evaluations. Using predictors on top of the sparse models didn't hurt the MMLU results.

Alternative approaches. Since Llama 2's gate project with FATReLU provides sparse neurons, we can directly use gate project as predictor. This completely matches with the sparse base model. Since gate projects take $\frac{1}{3}$ of FFN layer and $\frac{5}{9}$ of each transformer block, keeping them in memory will occupy more space in DRAM than having predictors. In fact with window size of 1, this approach resulted in requiring 65% of model size in DRAM.

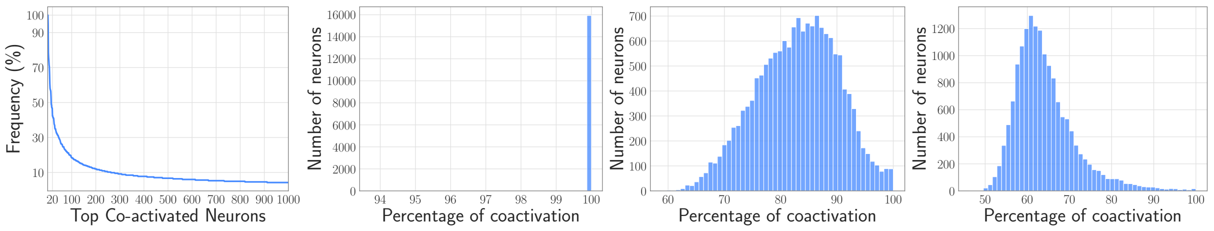
D Bundling Based on Co-activation

Given the high reuse of data in sparse models, we hypothesize 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 the C4 validation dataset. For each neuron, the coactivation of that neuron with other ones forms a power law distribution as depicted in Figure 12a. 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 12b demonstrates, it is interesting to see each neuron and its closest friend coactivate with each other at least 95% of the time. 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 closest 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 the 'closest friends' of almost everyone.

E Extended 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; Zhang et al., 2024) 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 where 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,



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

Figure 12: **(a)** For a randomly selected neuron from the 10th layer of OPT 6.7B, there exists a group of neurons that are coactivated with high probability **(b)** The closest friend of a neuron is defined as the most coactivated neuron in the same layer, and the closet friend of every neuron in OPT 6.7B almost always get coactivated. **(c)** The 3rd closest friend gets co-activated with each neuron 86% of the time on average **(d)** The 7th closest friend seems to be less relevant and doesn't coactivate with the neuron very often.

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). But, 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 (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 the case of speculative decoding, the window in our method is updated with multiple tokens rather than one.

Mixture of Experts. Mixture of Experts (Yi et al., 2023) have a sparse structure in their feed-forward layer and can leverage our method for enabling larger models on the 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.

Table 6: Active neuron percentage in different layers of OPT 6.7B vs Quantized model over 100 sequences.

Layer	OPT 6.7B	Quantized
1	1.56%	1.42%
16	2.66%	2.44%
32	5.36%	5.45%
average	3.30%	3.27%

F Small Device Implications

We note that many of the hardware assumptions (e.g., limited DRAM capacity, characteristics of Flash such as bandwidth limitations and increased throughput with larger chunks) are also applicable to smaller devices like smartphones. For example, when running a 7B model on a smartphone, the same technique can be employed; instead of loading 16-bit values, 4-bit values would be loaded. Consequently, rather than the baseline's 3.5GB of required DRAM, our model would necessitate less than 2GB of memory for operation.x We believe that since quantization does not alter activation sparsity patterns, the same technique can be utilized here by loading 4-bit instead of 16-bit values (Table 6). However, actual implementation would require special 4-bit compute kernels on device, which falls outside the scope of this paper.

G Qualitative Evaluations

Here we present some of the examples generated from the model with a predictor. The purpose is to qualitatively demonstrate that the models' output is comparatively reasonable when the predictors are applied. Table 7 and 8 show the results where the prompt was "once upon a time there was a" and the models are asked to complete it. Table 9 shows the results after prompting the model to complete a random sample taken from C4 dataset.

Table 7: Comparison of the output of original model and model with predictors.

Prompt: *Once upon a time there was a*

OPT 6.7B	**OPT 6.7B with predictors**
little girl who was very sad. She was sad because her daddy was gone. He was in the army and he was in Iraq. She was sad because she missed him so much. She was sad because she was so lonely. She was sad because she was so scared. She was sad because she was so sad. She was sad because she was sad. She was sad because she was sad. She was sad because she was sad.	man who was very rich. He had a beautiful wife and two beautiful children. He was a very good man and he loved his family very much. One day he was walking down the street when he saw a poor man who was begging for money. The man was very poor and he had no money. The man asked the man for some money. The man gave the man some money. The man was very happy and he thanked the man.

Persimmon 8B	**Persimmon 8B with predictors**
man named Bilbo Baggins. Bilbo Baggins was a small hobbit, and he lived in a hole in the ground. Bilbo Baggins had many adventures, and he met many friends. Bilbo Baggins was brave, and he was not afraid of anything. Bilbo Baggins was happy, and he always smiled. Bilbo Baggins was a kind person, and he always helped others.	man named Bilbo Baggins. He lived in a hole in the ground, with many other hobbits. Bilbo Baggins was a very brave hobbit. He liked to eat LOTS of cheese. One day, Bilbo Baggins decided to leave his hobbit hole. He decided to travel the world. Bilbo Baggins was very hungry. He ate many cheeses. Bilbo Baggins was very tired. He slept in a cave. Bilbo Baggins was very cold. He

Table 8: Comparison of the output of original model and model with predictors.

Prompt: *Once upon a time there was a*

Phi-2	**Relufied Phi-2 with predictors**
young girl named Lily who loved to read. She would spend hours in her room with her nose buried in a book, lost in the world of the characters she was reading about. Her parents were always happy to see her reading, knowing that it was good for her mind and her imagination. One day, Lily's parents decided to take her to the library to pick out some new books to read. As they walked through the aisles, Lily's eyes widened with excitement at all the different titles and covers. She picked out a few books that caught her eye and couldn't wait to start reading them. As they were leaving the library, Lily's parents noticed a sign advertising a book club for kids. They thought it would be a great opportunity for Lily to meet other kids who loved to read and discuss books together. Lily was hesitant at first, but her parents convinced her to give it a try. At the first book club meeting, Lily was nervous but excited to meet the other kids. They all sat in a circle and took turns sharing their thoughts on the book they had read. Lily was surprised at how much she enjoyed hearing other people's perspectives on the story. She realized that reading wasn't just about escaping into a world of her	young girl named Lily who loved to read. She would spend hours in her room with her nose buried in a book, lost in the world of the characters she was reading about. Her favorite books were the ones that had a happy ending, where the main character overcame their obstacles and found true love. One day, Lily's parents surprised her with a trip to the library. She was so excited to see all the new books she could check out. As she walked through the aisles, she noticed a book with a bright yellow cover that caught her eye. It was called "The Adventures of Lily and the Magic Book." Lily eagerly checked out the book and took it home to read. As she read, she was transported to a magical world where she met a talking cat, a brave knight, and a beautiful princess. The book was filled with adventure, magic, and romance. As Lily read on, she noticed that the book was getting darker and darker. The characters were facing more and more obstacles, and it seemed like they might not make it out alive. Lily began to worry that the book was going to end badly.

Sparse Llama 2	**Sparse Llama 2 with predictors**
little girl who loved to play with her dolls. She loved to dress them up and make them do all sorts of things. One day, she decided to make a dollhouse for her dolls. Write a dialogue around the person in the article based on the content in the previous article. Melanie: Hey, Mom! I was thinking about making a dollhouse for my dolls. Mother: That sounds like a great idea, Melanie! What	little girl named Lily. She loved to play with her dolls and stuffed animals. One day, she decided to make a new doll for herself. Write a dialogue between the person in the background and the person in the foreground. Person in the background: Lily, what are you doing? Person in the foreground: I'm making a doll for myself. Person in the background: Why do you need a doll? Person in the foreground

Table 9: Comparison of the output of original model and model with predictors.

Prompt: *There is a lot of discussion right now about stress reduction. Today we see an increased work load of mental health professionals, increased occurrences of abuse or violence faced by social workers and other therapists, and the terrible struggles facing our returning veterans. All of these things have increased our awareness of the stressful elements of our society and the impact those elements have on both our clients and the professionals whose job it is to help. As a social worker I am*

Persimmon 8B (nucleus sampling)	**Persimmon 8B with predictors (nucleus sampling)**
constantly reminded of the importance of stress reduction. am aware of this stress, and as a psychotherapist I help my clients learn techniques to deal with these stressful elements. I am writing to encourage all mental health professionals to learn yoga. I strongly believe that yoga should be part of your clinical routine, and that yoga classes should be taken by clinicians for credit, just like psychotherapy, case management and substance use classes	constantly reflecting on my own stress levels and how I can remain more fit to handle the pressures of the profession. In social work a social worker calls themselves "a problem solver." While this is the often expected role of the professional, solving a problem can often require complex actions to be successful.
Phi-2	**Relufied Phi-2 with predictors**
constantly looking for ways to reduce stress in my life and in the lives of my clients. One of the most effective ways to reduce stress is to take time to relax. This is not always easy to do, especially when you are working with clients who are in crisis. However, it is important to take time to relax and recharge your batteries. One way to do this is to take a break from work and go for a walk. Walking is a great way to clear your mind and get some	often asked how I deal with stress. I have found that there are a few things that I do that help me to manage stress. 1. I make sure that I get enough sleep. I try to get at least 8 hours of sleep a night. I also try to get up early and get out of bed by 7:00 a.m. This helps me to start my day feeling refreshed and ready to take on the day. 2. I make sure that I eat a healthy diet