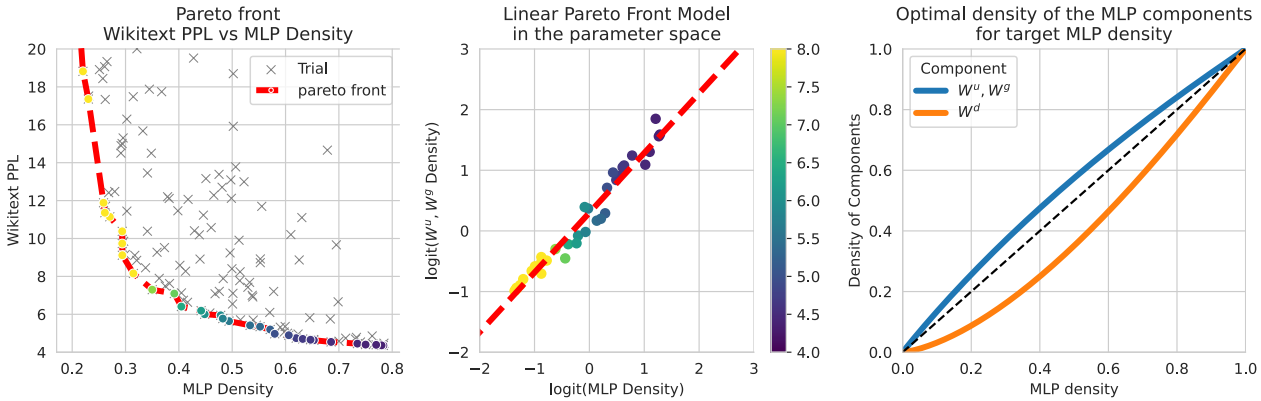

Figure 12. Optimal allocation of weight density between $\widetilde{\boldsymbol{W}}^d$ and $(\widetilde{\boldsymbol{W}}^u, \widetilde{\boldsymbol{W}}^g)$.

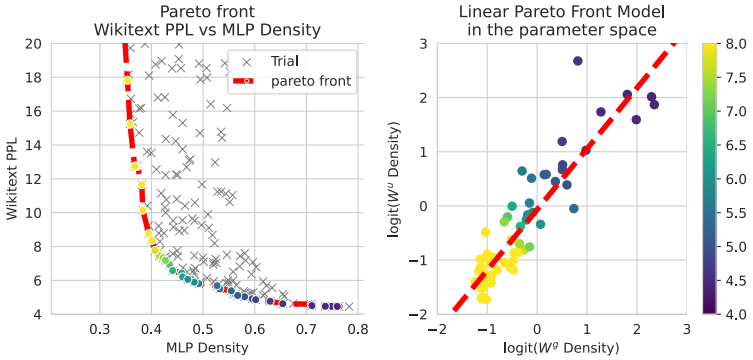

Figure 13. Allocation of weight density between $\widetilde{\boldsymbol{W}}^u$ and $\widetilde{\boldsymbol{W}}^g$ matrices for $\widetilde{\boldsymbol{W}}^d$ with fixed density of 50% for *phi-3-medium model* on the Wikitext dataset.

A HW Simulator description

The main parameters needed for simulation are therefore the DRAM capacity, and Flash and DRAM read/write speeds. We notice similar specifications in the processors for latest popular smartphone devices: Apple A18 (Wikipedia, 2024a) and Snapdragon® 8s Gen 3 SM8635 platform (Wikipedia, 2024b). While latest UFS versions can reach up to 5.8 GB/s in bandwitdh (Wikipedia, 2024), the effective reading speeds might vary significantly depending on hardware interfaces and whether sequential or random reads are prevalent (Xue et al., 2024). The same applies for NVMe storage systems adopted by Apple devices (Labaran, 2023). Unless otherwise stated, results throughout the paper are based on simulations for Apple A18 with DRAM I/O speed of 60 GB/s, and Flash read speed of 1 GB/s.

A significant part of the available DRAM in smartphones is usually reserved by the OS or other applications (Danielson, 2023). We consider multiple values for available DRAM capacity to investigate how differently sized LLMs perform in memory-constrained scenarios. We include ablations with varying DRAM capacity (Table 6) and Flash read speed (Table 7) to assess whether the performance is consistent over several device specifications.

Consistently with Xue et al. (2024), for cache simulation, we statically allocate to DRAM all the layers which do not undergo dynamic pruning, such as attention layers, embeddings, but also KV-cache and for certain methods, auxiliary modules like predictors. Since these layers are needed to process each token, pre-loading them in DRAM is an optimal choice (Xue et al., 2024). We then allocate the remaining DRAM capacity uniformly to all the MLP layers to enable the dynamic loading and caching of MLP weights. We did not find significant improvements when exploring non-uniform cache allocation.

B Hyperparameter tuning

B.1 Density of up, down and gate matrices

We determined the optimal memory allocation for DIP in a three steps procedure illustrated in Figure 13:

1. Run a 2D optimization on Memory vs Perplexity and

Table 3. Experimental results for dynamic sparsity methods at 60% MLP density.

	WikiText-2 (Perplexity ↓)				MMLU (5-shot accuracy ↑)			
	Phi3Med	Phi3Mini	Llama8B	Mistral7B	Phi3Med	Phi3Mini	Llama8B	Mistral7B
Dense	4.29	6.01	6.14	5.25	78.14	70.62	65.30	62.68
GLU Pruning (oracle)	4.35	6.04	6.20	5.26	78.28	70.42	65.00	62.35
SparseGPT (unstructured)	5.08	6.63	6.87	5.46	76.26	66.81	60.91	61.23
Gate Pruning	6.36	7.97	10.22	8.67	74.85	62.14	51.62	55.53
Up Pruning	5.68	7.51	8.50	5.74	75.54	64.67	60.29	59.91
DejaVu	5.69	7.74	8.35	5.87	72.15	59.34	55.30	57.58
CATS	5.91	7.43	9.47	8.89	75.91	64.63	55.11	55.65
CATS+LoRA	4.95	6.65	8.61	6.63	75.79	65.64	53.94	56.44
DIP	4.85	6.4	6.66	5.39	77.29	68.88	63.17	61.19
DIP+LoRA	4.62	6.35	6.63	5.38	77.39	68.77	62.95	61.50

Table 4. Experimental results for dynamic sparsity methods at 40% MLP density.

	WikiText-2 (Perplexity ↓)				MMLU (5-shot accuracy ↑)			
	Phi3Med	Phi3Mini	Llama8B	Mistral7B	Phi3Med	Phi3Mini	Llama8B	Mistral7B
Dense	4.29	6.01	6.14	5.25	78.14	70.62	65.30	62.68
GLU Pruning (oracle)	4.64	6.24	6.52	5.35	77.74	69.23	64.28	61.67
SparseGPT (unstructured)	6.53	9.51	9.68	6.92	67.79	53.01	48.51	50.15
Gate Pruning	550.27	496.34	>1000	>1000	29.45	26.17	24.51	24.75
Up Pruning	18.57	63.59	68.15	20.44	52.09	29.94	25.32	30.37
DejaVu	6.82	10.55	11.25	6.83	64.77	49.41	41.5	49.38
CATS	196.11	122.93	>1000	>1000	34.8	27.89	25.25	25.12
CATS+LoRA	6.90	9.56	787.70	20.33	60.05	44.67	23.63	26.31
DIP	6.5	8.66	9.01	6.16	72.95	60.33	53.72	55.92
DIP+LoRA	5.64	7.68	8.71	6.03	72.43	61.14	54.35	56.20

determining the pareto optimal configurations

2. Modelling the pareto optimal solutions in the parameter space. Here we considered linear models from the target MLP density and the density of the up and gate matrices in logit space.

3. Use the fitted model to determine the optimal allocation for a target MLP density.

Using the same procedure, we determined there is no significant gain in using different sparsity levels for the Up $\boldsymbol{W}^u$ and Gate $\boldsymbol{W}^g$ matrices (Figure 13). We found that the optimal allocation is consistent across the tested LLMs.

C Additional Results

We show in Figure 14 additional results in terms of perplexity and MLP sparsity for Phi-3-Mini, Llama-v3-8B and Mistral-7B, complementing the study on Phi-3-Medium presented in Section 6.2. Table 4 and Table 3 report the perplexity and accuracy values evaluated at 60% and 40% MLP density respectively. Consistently to the results in Table 1, we don't consider memory overhead introduced by different methods. The MLP density for CATS and CATS+LoRA may vary up to 2% from the operating point since the fixed estimated threshold may result in slightly different sparsity levels when evaluated on different datasets.

Table 5 compare the performance of the same models at 50% MLP sparsity on a wide range of tasks including common reasonong (ARC, BoolQ, HellaSwag, PIQA, Wingogrande), multi-lingual reasonong (MGSM) and language understanding (MMLU-Pro). The evaluation procedures follows the protocol described in the *Language Model Evaluation Harness* (Gao et al., 2024). MGSM and MMLU-Pro require generation of multiple tokens and the corresponding accuracy is evaluated by considering exact matches. The reported values of MGSM refer to the average performance in 11 languages.

These additional results reinforce the trend we observed for perplexity and MMLU in Table 1, with DIP outperforming unstructured SparseGPT, DejaVU and CATS on most tasks and architectures.

Table 5. Accuracy at 50% MLP sparsity on various language reasoning and understanding tasks.

	Accuracy ↑							
	ARC (easy)	ARC (challenge)	BoolQ	HellaSwag	PIQA	Winogrande	MGSM	MMLU-Pro
Phi3Med								
Dense	85.40	61.43	88.53	65.07	80.85	76.56	32.95	52.69
GLU pruning (oracle)	85.31	60.58	88.75	65.09	81.39	75.37	33.24	52.19
SparseGPT (unstructured)	84.05	55.38	88.93	60.83	78.78	75.22	26.80	44.13
DejaVu	83.33	54.52	87.74	59.90	77.86	74.82	20.11	37.02
CATS	80.22	54.78	86.18	60.73	74.48	66.22	13.60	38.50
DIP	81.06	56.14	88.69	62.33	76.77	72.22	33.24	48.01
Phi3Mini								
Dense	81.99	53.75	85.17	59.01	80.41	73.48	32.87	44.46
GLU pruning (oracle)	82.24	55.12	84.68	59.15	79.71	73.64	31.85	43.42
SparseGPT (unstructured)	78.28	47.44	80.61	53.92	77.26	70.64	20.22	30.30
DejaVu	75.29	45.14	74.59	43.74	76.50	62.98	6.33	24.17
CATS	75.55	45.65	65.99	52.50	72.91	62.12	6.40	21.76
DIP	79.42	53.33	83.15	57.22	77.31	68.43	26.36	36.66
Llama8B								
Dense	80.09	50.06	81.35	60.19	79.71	72.38	9.75	34.47
GLU prining (oracle)	79.38	49.83	80.52	60.21	79.82	73.24	10.07	33.69
SparseGPT (unstructured)	75.42	41.30	78.99	54.58	77.09	71.11	5.13	24.00
DejaVu	67.59	39.76	74.89	53.89	76.44	68.59	3.27	20.10
CATS	54.38	32.25	56.70	45.43	70.24	57.54	0.47	7.65
DIP	77.57	46.50	79.51	57.90	77.53	71.11	7.35	29.48
Mistral7B								
Dense	80.89	50.43	83.61	61.21	80.58	73.88	4.40	30.39
GLU pruning (oracle)	80.51	50.26	83.33	61.59	80.63	73.09	4.47	30.18
SparseGPT (unstructured)	76.68	44.88	76.24	57.05	78.18	71.82	4.11	24.19
DejaVu	77.40	45.90	79.24	60.47	79.33	72.22	2.87	23.55
CATS	51.56	32.00	73.85	49.67	70.02	61.64	0.47	5.66
DIP	79.12	48.98	83.15	60.70	79.54	72.53	4.29	28.02

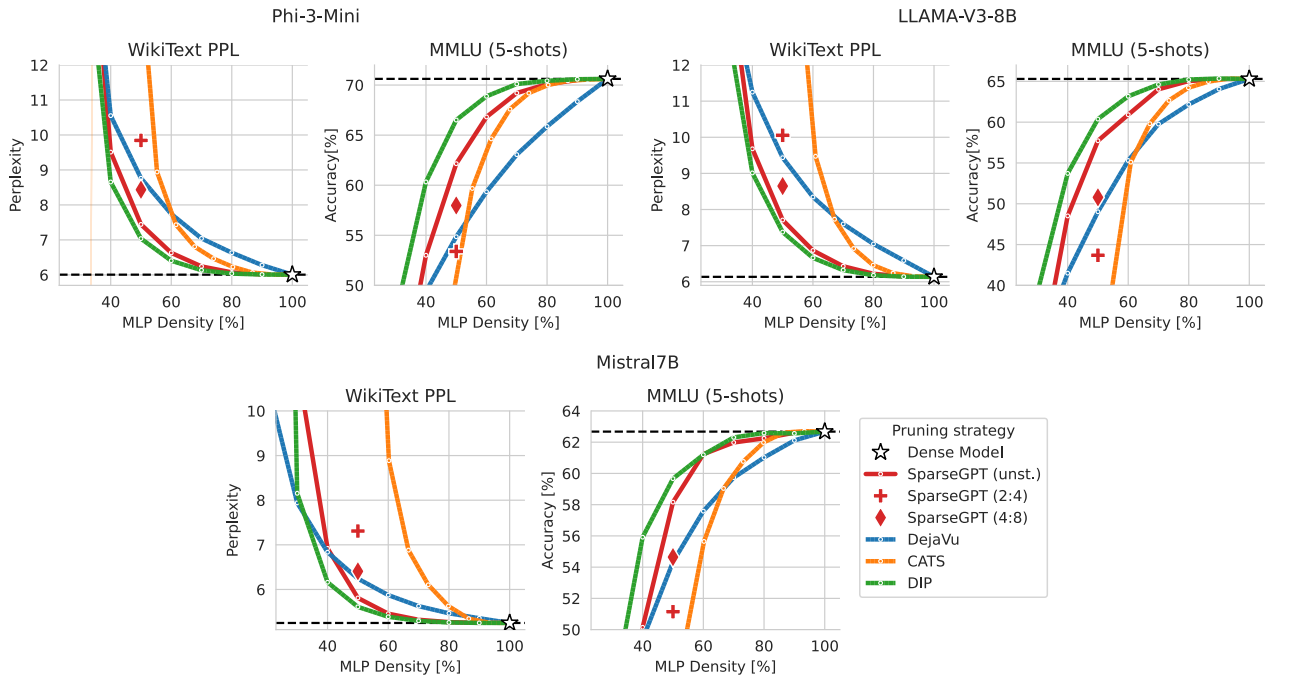

Figure 14. Downstream WikiText Perplexity and MMLU accuracy as a function of MLP Density for Phi-3-Mini, LLama-v3-8B, and Mistral-7B models.

Table 6. Comparison of throughput for dynamic sparsity methods at different DRAM sizes with Phi-3-Medium quantized to 4 bits. We report the highest throughput achieved at a 0.5 increase in perplexity on WikiText-2 over the dense model.

DRAM size	2 GB	4 GB	6 GB
	Throughput [tok/s] ↑		
Dense	0.19	0.29	0.71
GLU Pruning	0.24	0.45	1.83
Up Pruning	0.27	0.52	1.35
CATS	0.25	0.47	1.21
DIP-CA	0.31	0.56	1.94

D Ablation on hardware specifications

For main experiments we considered a fixed hardware setting in line with the specifications of Apple A18 processors. We now consider different target devices and scenarios by simulating changes in DRAM size and Flash read speed.

DRAM size In Table 6 we consider use-cases with lower DRAM availability at 2GB, simulating the use-case of budget smartphones, and higher DRAM availability at 6GB, which represents high-end devices, or scenarios where the OS and background applications require less memory. In all scenarios, DIP yields better throughput at a fixed perplexity increase of 0.5 over the baseline. Remarkably, DIP improves throughput by 170% against the dense model with 6GB of DRAM. This stems from the increased availability in caching space for the linear layer, which increases the cache hit rate to 89%, compared to the 53% at 4GB and only 8% at 2GB. At even higher DRAM sizes, where the dense model fully fits in cache, we expect GLU Pruning to outperform DIP-CA, as the overhead in loading more weights from DRAM (for GLU Pruning) has a minor impact with respect to the loss in accuracy resulting from sparsifying all MLP layers (in DIP-CA).

Table 7. Comparison of throughput for dynamic sparsity methods at different Flash reading speeds with Phi-3-Medium quantized to 4 bits. We report the highest throughput achieved at a 0.5 increase in perplexity on WikiText-2 over the dense model.

Flash read speed	0.5 GB/s	1 GB/s	2 GB/s
	Throughput [tok/s] ↑		
Dense	0.15	0.29	0.59
GLU Pruning	0.23	0.45	0.91
Up Pruning	0.26	0.52	1.01
CATS	0.24	0.47	0.91
DIP-CA	0.28	0.56	1.09

Flash reading speed We evaluate DIP against previous methods at different operating points in terms of Flash read-

ing speed. The actual transfer speed can change significantly depending on storage type, hardware interfaces, and data representation for the saved models. While takeaways and relative improvements at different reading speeds do not change, the results in Table 7 highlight how the absolute throughput values change almost at the same rate as the increases in Flash reading speed, confirming that this is the main bottleneck to achieve high-latency in memory-constrained scenarios.

E Artifact Appendix

E.1 Abstract

This Artifact Appendix describes the experimental workflow, artifacts and results from this paper submitted for the Artifact Evaluation at MLSys 2025. The experiments described in this paper can be reproduced through the open source code released as a code artifact (Federici et al., 2025). The code should be executed in a Python 3 environment as described in this appendix. The setup relies on GPU hardware to accelerate the experiments, with the smallest model requiring 40GB of VRAM. Datasets and models used in this paper can be obtained from Huggingface[4].

E.2 Artifact check-list (meta-information)

- **Algorithm:** Efficient LLM inference.
- **Models:** Phi-3-Medium, Phi-3-Mini, Llama-v3-8B, Mistral-v01-7B.
- **Data sets:** Wikitext, MMLU, SlimPajama (optional).
- **Run-time environment:** Python 3.8 and above interpreter and latest pip version to install the required packages.
- **Hardware:** A torch-enabled GPU with 80GB of VRAM is required for most models. For Phi-3-Mini, 40GB of VRAM are enough.
- **Execution:** *python scripts/run_experiment.py –help*
- **Metrics:** Perplexity, 5-shot accuracy, throughput, memory footprint, MLP density.
- **Output:** Metrics for a given method, model, dataset and hardware simulator setup.
- **Experiments:** Experiments to reproduce Table 1 (perplexity and 5-shot accuracy at 50% sparsity) and Table 2 (throughput at given perplexity increase) in the paper.
- **How much disk space required (approximately)?:** Less than 1 MB for the code, Up to 1GB for the pyhon environment setup. Up to 100GB for the LLM models.
- **How much time is needed to prepare workflow (approximately)?:** A few minutes for the environment setup and download each model and dataset.
- **How much time is needed to complete experiments (approximately)?:** Less than 10 minutes for a single run with Phi-3-Mini on Wikitext. More than 200 GPU hours to reproduce all experiments.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** BSD 3-Clause Clear License.
- **Data licenses (if publicly available)?:** As specified for each Huggingface dataset.
- **Workflow framework used?:** No.
- **Archived (provide DOI)?:** 10.5281/zenodo.15088634

[4] https://huggingface.co/models

E.3 Description

E.3.1 How delivered

Our source code and instructions for installation, setup and experiment reproduction are publicly available at: https://github.com/Qualcomm-AI-research/dynamic-sparsity.

E.3.2 Hardware dependencies

We recommend testing on a Linux machine with torch-enabled GPU. We used a x86_64 architecture with an AMD 64-core processor, Nvidia H100 with 80GB of VRAM and CUDA 12.2.

E.3.3 Software dependencies

Python 3.8 and above interpreter and latest pip version to install the required packages. Tested with Python 3.8 and pip 24.3.1. Packages are listed in *requirements.txt*.

E.3.4 Data sets

The Wikitext and MMLU data sets are required to reproduce the main experiments, and SlimPajama is needed when running the method variation employing LoRA finetuning.

E.4 Installation

Python 3 environment with installed packages as listed in *requirements.txt*. Dataset and models must be downloaded from Huggingface. All details on installation and setup are provided in the "Getting Started" section of the README documentation.

E.5 Experiment workflow

In the "Usage" subsection of the README file we explain how to run experiments with the released code, including a description of all the main parameters regulating an experiment.

E.6 Evaluation and expected result

In the "Reproducing Results" section, we explain how to prepare and launch the experiments to reproduce the results in Table 1 (perplexity and 5-shot accuracy at 50% sparsity) and Table 2 (throughput at given perplexity increase) in the paper.

E.7 Experiment customization

The provided source code implements a set of dynamic sparsity methods for efficient LLM inference. We showcase in our paper experimental results on a selected set of data sets and models, but experimenting with different models and data sets is also possible with minimal code changes. Similarly, we release in our source code a hardware simulator to assess the system throughput under different configurations. These configurations are flexible and can be easily extended to simulate new scenarios. As an example, we included in */scripts/config/hw_simulator/processor* the configurations for a variety of Apple processors based on the specifications provided in Wikipedia. An overview of the main components in our codebase is included in the "Repository Structure" section.