Characterization of Unstructured Sparsification of Transformer Language Models

Tahar HERRI
McMaster University
ECE 718 — Compiler Design for HPC
herrit@mcmaster.ca

December 2025

Abstract

This report presents a controlled study of *unstructured weight pruning* in decoder-only Transformer language models under strict hardware constraints. We compare three variants of the same base model: a dense reference, a masked pruned model where pruned weights are set to zero but still executed with dense kernels, and a CSR variant where pruned linear layers are stored in Compressed Sparse Row (CSR) format and executed with a sparse-aware forward. Instead of relying on large-scale generation benchmarks, we combine (i) **structural metrics** (parameter counts, sparsity, memory footprint, FLOPs proxies) with (ii) a **logits-based behavioural proxy** (*top-1 agreement* with the dense model). Across our runs, CSR makes sparsity *physical* (reducing storage and theoretical compute in pruned layers), while behavioural changes are driven primarily by pruning rather than by the sparse representation.

1 Introduction

Decoder-only Transformer language models (LMs) such as OPT and Pythia rely heavily on large linear operators evaluated at every token, making inference cost scale with model size and sequence length [1, 2]. Sparsification addresses this cost by reducing the number of active parameters or operations, typically via pruning [3, 4]. In this report, we focus on **unstructured magnitude pruning** (zeroing individual weights based on absolute value), following the classical pruning framing introduced in early work [3] and surveyed in later analyses [4].

A key practical caveat is that **unstructured sparsity does not automatically yield speedups**: irregular masks require sparse representations and sparse-aware kernels to translate zeros into reduced work [4, 7]. Therefore, the purpose of this project is *not* to reach state-of-the-art accuracy under pruning (as in one-shot LLM pruning methods like SparseGPT or Wanda [5, 6]), but to build a **reproducible analysis pipeline** that makes the following two questions measurable under limited hardware:

1. **Behaviour:** how much does pruning change token-level predictions compared to the dense reference?
2. **Structure:** where do parameters, sparsity, and compute proxies concentrate inside the architecture?

2 Scope and Experimental Setup

2.1 Models

Experiments use decoder-only Transformer LMs, selected based on what could be loaded and manipulated on available hardware:

- **OPT-125M** as the main model for the full CPU-friendly analysis [1].
- **Pythia-410M** for limited GPU-based tests when possible [2].

Both are decoder-only autoregressive Transformers, suitable for controlled token-level comparisons [1, 2].

2.2 Compared Variants

For a fixed pretrained checkpoint, three variants are compared:

- **Dense**: original model parameters.
- **Masked 30%**: unstructured magnitude pruning applied to selected linear layers; pruned weights are set to zero but the weight tensors remain dense (dense execution).
- **CSR 30%**: identical pruning mask, followed by conversion of selected pruned linear layers to a CSR representation and execution via a sparse-aware forward (sparse execution path).

This Dense/Masked/CSR split explicitly separates "zeros in memory" from "zeros exploited by execution" [4, 7].

2.3 Which Layers Are Pruned (and Why)

Figure 1 summarizes the compute structure of a decoder-only Transformer block.

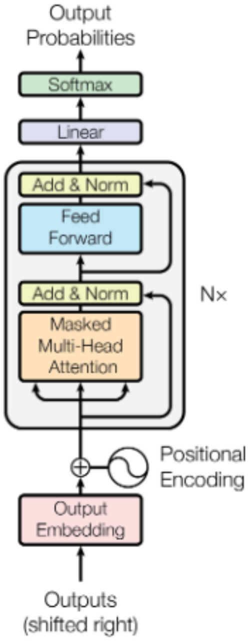


Figure 1: Decoder-only Transformer block (repeated $N\times$). The study targets the largest **Linear** maps in attention and MLP(feed-forward) sub-layers, which dominate parameter count and matrix-multiply cost in practice.

Pruned layers (dominant linear operators). We apply unstructured magnitude pruning to the main dense linear maps:

- **Attention projections:** W_Q, W_K, W_V and the attention output projection W_O.
- **MLP/feed-forward:** the two linear layers surrounding the non-linearity (often `fc1`, `fc2`).

These layers are repeated across blocks and are the primary GEMM-like compute sites in decoder-only Transformers [1, 2].

Layers kept dense (methodological choice). We keep embeddings, the final output projection (`lm_head`), and normalization layers dense. This is not imposed by PyTorch: it is a deliberate design choice to keep the pipeline stable and interpretable. Conceptually, pruning can be applied broadly [3, 4], but in this didactic study we isolate pruning to the main linear compute blocks while leaving (i) *fragile* components (embeddings, output head) and (ii) *low-parameter* components (norms) unchanged to reduce confounders.

2.4 Hardware Constraints

Pruning requires access to model weights, meaning the full model must be loaded to perform pruning, masking, and (for CSR) conversion. In practice, limited RAM/VRAM restricted the size of models that could be loaded and evaluated, and prevented large-scale generation or perplexity benchmarks. For that reason, we emphasize **lightweight, controlled metrics** that can run deterministically on CPU: parameter statistics, group-wise FLOPs proxies, and logits-level behavioural similarity.

2.5 Implementation

All experiments are executed from the project notebooks, which call into the repository source code under `src/`. In practice, the notebooks orchestrate the workflow (Dense → Masked → CSR) and generate the figures, while the underlying operations are implemented as reusable modules:

- `src/pruning/`: loading checkpoints and applying unstructured magnitude pruning (mask + zeroing) on selected linear layers;
- `src/wrappers/`: CSR conversion and sparse-aware forward replacements for selected layers;
- `src/eval/`: metric computation and analysis utilities used to produce the reported plots.

Reproducibility conventions (seeds, device selection, and run notes) are documented in the repository README, and all variants are evaluated on identical tokenized inputs.

The full implementation, including notebooks and source code, is available at: https://github.com/TaharHERRI/Edu-Sparsify-LLMs.git.

3 Metrics: Definitions, Interpretation, and How They Connect

The analysis is built as a chain:

$$\text{(where sparsity appears)} \rightarrow \text{(what changes structurally)} \rightarrow \text{(how behaviour drifts)}$$

This section defines each metric and clarifies what it can and cannot claim, following the broader sparsity literature [4].

What is a proxy metric. In this report, a *proxy* denotes a measurable quantity that is not the target metric of ultimate interest, but that correlates with it and can be evaluated reliably under the given constraints. Proxy metrics are used when direct measurement is impractical or too costly (e.g., perplexity or wall-clock latency), and are interpreted comparatively rather than as absolute performance indicators.

3.1 Global Sparsity (Zeros in Stored Parameters)

Let T be the total number of scalar parameters and N the number of *non-zero* parameters. Global sparsity is:

$$\text{sparsity} = 1 - \frac{N}{T}. \tag{1}$$

This measures the fraction of weights that are exactly zero *in the stored representation*. Because pruning is applied only to a subset of layers, the *global* sparsity is necessarily below the nominal pruning ratio (e.g., "30% pruning").

Why this matters for interpretation. Global sparsity is easy to compute, but it mixes pruned and unpruned parts. To understand *where* sparsity lives, we complement it with group-level sparsity (Section 3.4) and see Figure 2b and Figure 3.

3.2 Model Size (Memory Footprint of the Representation)

Model size estimates the memory footprint of stored parameters:

$$\text{size\_MB} \approx \frac{(\text{number of stored values}) \times (\text{bytes/value})}{1024^2}.$$

CSR accounting details. For a CSR-stored linear layer, the reported size includes: (i) non-zero values, (ii) column indices for each non-zero, and (iii) row pointers of length (rows + 1). Unless otherwise stated, indices and pointers are assumed to use 32-bit integers.

This metric answers whether sparsity becomes *physical* savings in storage. With Masked pruning, zeros are stored inside dense tensors, so size typically remains unchanged. With CSR, zeros in converted layers are not stored explicitly, so size can drop significantly [4].

3.3 Top-1 Agreement (Logits-Based Behavioural Proxy)

Full quality evaluation of LMs usually relies on perplexity or downstream tasks. When that is infeasible, a *proxy* can still quantify how pruning perturbs local predictions on a fixed corpus. We therefore use **top-1 agreement** as a deterministic, GPU-free behavioural proxy.

Definition. Given dense reference logits $\ell^{(d)} \in \mathbb{R}^{B \times L \times V}$ and variant logits $\ell^{(v)}$ computed on the same tokenized inputs, top-1 agreement is:

$$\text{top1\_match} = \frac{\#\{(b,t) : \arg\max_k \ \ell^{(d)}_{btk} = \arg\max_k \ \ell^{(v)}_{btk}\}}{\#\{\text{valid token positions}\}}. \tag{2}$$

Because $\arg\max(\text{softmax}(\ell)) = \arg\max(\ell)$, the comparison can be performed directly on logits without applying softmax.

Interpretation at the logits level. Top-1 agreement compares model predictions at the level of logits, before any sampling or decoding heuristic is applied. Because the argmax of the softmax distribution is identical to the argmax of the raw logits, the comparison can be performed directly on logits without loss of correctness. A detailed explanation of the logits–softmax–argmax pipeline is provided in Appendix A.

What top-1 agreement captures. Top-1 agreement measures *token-level decision stability*: on how many positions the most likely token ID remains the same after pruning. It is a strict notion of behavioural similarity: a small logit change can flip the argmax in ambiguous positions.

What top-1 agreement does *not* capture (critical context). Top-1 agreement is not a replacement for perplexity or human-perceived generation quality. In particular:

- It ignores **confidence margins**: the top-1 token may match while probability mass shifts substantially.
- It ignores **rank structure**: the top-2/top-5 sets may change without affecting top-1.
- It is **corpus-dependent**: a small evaluation set can miss rare phenomena.
- It is **local**: it does not directly assess long-range generation coherence.

For these reasons, we interpret top-1 agreement only *in conjunction with* structural metrics: group-level sparsity explains *what* was removed, and top-1 agreement quantifies *how* local predictions react to that removal.

3.4 Group-Level Structural Metrics (Where Parameters and Sparsity Live)

To connect global statistics to architecture, each module is assigned to a group: `embedding`, `attention_linear`, `mlp_linear`, `lm_head`, `norm`. For each group g:

$$\text{sparsity}_g = 1 - \frac{N_g}{T_g}, \tag{3}$$

$$\text{param\_frac}_g = \frac{T_g}{\sum_{g'} T_{g'}}. \tag{4}$$

FLOPs proxy per token (dense vs sparse). For a dense linear map with weight matrix $W \in \mathbb{R}^{\text{out} \times \text{in}}$ applied as a matrix-vector product, a standard proxy is:

$$\text{FLOPs}_{\text{dense}} \approx 2 \cdot \text{in} \cdot \text{out}.$$

For a sparse representation with $\text{nnz}(W)$ non-zeros:

$$\text{FLOPs}_{\text{sparse}} \approx 2 \cdot \text{nnz}(W).$$

Limitations of the FLOPs proxy. The FLOPs proxy is intentionally optimistic. Sparse kernels incur additional overheads due to irregular memory access, index indirection, and reduced arithmetic intensity. As a result, a reduction in FLOPs proxy does not guarantee a proportional reduction in wall-clock latency, especially on CPUs or non-specialized hardware.

This is a proxy—not a measured latency—but it matches the key sparsity principle: to convert zeros into less arithmetic, kernels must scale with non-zeros [4, 7]. Accordingly, the FLOPs proxy changes for CSR (sparse-aware forward) but not necessarily for Masked (dense kernels still multiply zeros).

4 Notebook 1: Global Behaviour (Size, Sparsity, Top-1 Agreement)

Notebook 1 reports global metrics for Dense, Masked 30%, and CSR 30%. Figure 2 should be read as a triplet: memory footprint (Figure 2a), global sparsity (Figure 2b), and behavioural drift via top-1 agreement (Figure 2c).

Memory footprint (Figure 2a). Dense and Masked have essentially identical size: masking does not change tensor storage. CSR is smaller because zeros in converted layers are not stored explicitly.

Global sparsity (Figure 2b). Global sparsity is well-defined only for representations that explicitly store zeros. For Dense and Masked models, masking increases global sparsity, although it remains below the nominal 30% target because embeddings, the output head, and normalization layers are kept dense. For the CSR variant, zeros are not stored by construction, so a sparsity ratio is not reported. To understand *where* sparsity lives inside the architecture, we therefore use *group-level sparsity* on the Masked model (Figure 3), which isolates the pruned groups from the dense components.

Behavioural proxy (Figure 2c). Masked and CSR obtain the same top-1 agreement in the shown run. This supports a key separation:

- **Pruning** (weight removal) drives behavioural changes.
- **CSR** primarily changes storage/execution, and does not add additional behavioural drift beyond the mask (up to minor numerical effects).

This is consistent with the idea that different sparse representations affect efficiency more than the underlying function when they implement the same sparsity pattern [4].

5 Notebook 2: Structural Analysis (Where Sparsity and Compute Live)

Notebook 2 explains *why* the global metrics behave as they do by decomposing sparsity and parameter mass into groups.

5.1 Where Sparsity Appears (By Group)

Figure 3 directly localizes sparsity: it concentrates in `attention_linear` and `mlp_linear`, the only groups we prune. Unpruned groups (embeddings, `lm_head`, norms) remain dense by design.

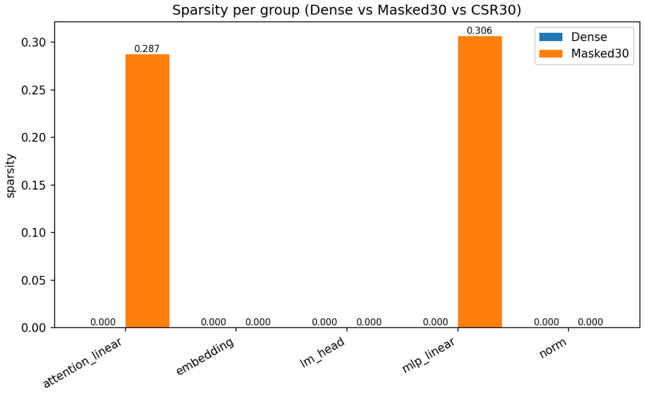


Figure 3: Group sparsity (Dense vs Masked30 vs CSR30). Sparsity appears where pruning is applied.

5.2 Where Parameters Are Concentrated (By Group)

Figure 4 shows how parameters are distributed across major groups. Importantly, **pruning does not change the model's computational graph**: the same layers are executed in the same order, with the same shapes. Unstructured sparsification only changes the *numerical values* of weights (setting some to zero), and CSR additionally changes how some weights are *stored*. As a result, the **parameter layout by group is the same** for Dense, Masked, and CSR: pruning does not move parameters between groups.

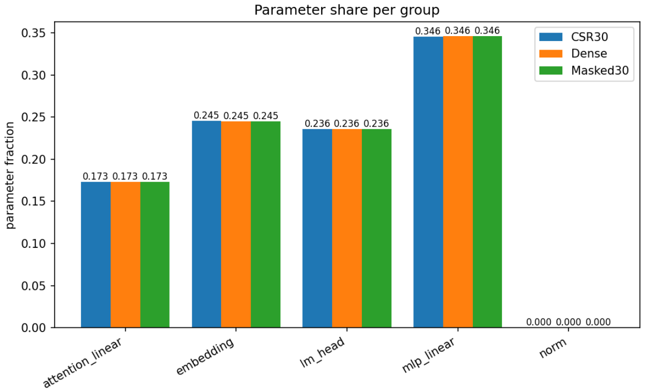
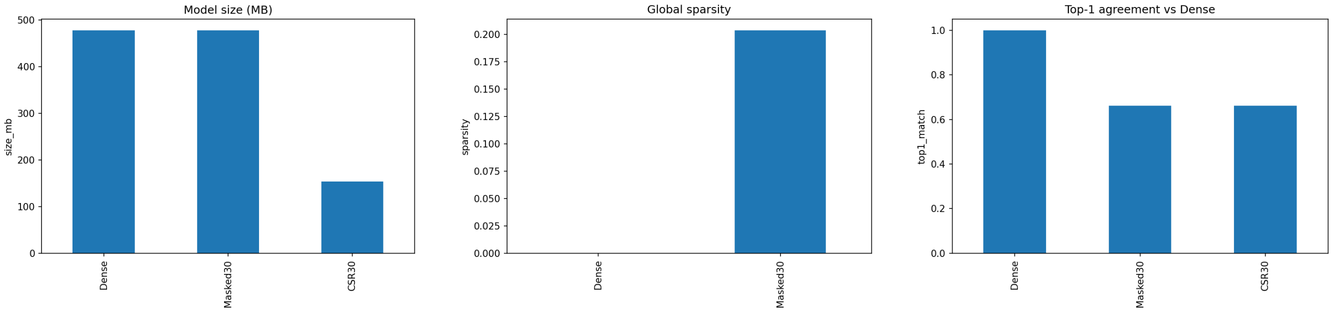


Figure 4: Parameter fraction per group (identical across variants). Large dense groups (e.g., embeddings, `lm_head`) cap end-to-end compression.

Connection to global size. Because embeddings and `lm_head` represent a large fraction of parameters and are kept dense, they limit how much total size can drop even when pruned linear layers compress strongly (Figure 2a vs Figure 4). In other words, CSR can shrink the groups it targets, but it cannot shrink the dense groups that remain unchanged.



(a) Model size (MB).

(b) Global sparsity.

(c) Top-1 agreement vs Dense.

Figure 2: Global metrics for Dense, Masked 30%, and CSR 30% (OPT-125M run shown).

5.3 Where Theoretical Compute Lives (FLOPs Proxy)

Figure 5 links sparsity to a compute proxy. Dense and Masked match because dense kernels still execute multiplications on zeros. CSR reduces the FLOPs proxy in pruned linear groups because its sparse-aware forward scales with non-zeros, reflecting the core requirement for speedups under irregular sparsity [4, 7].

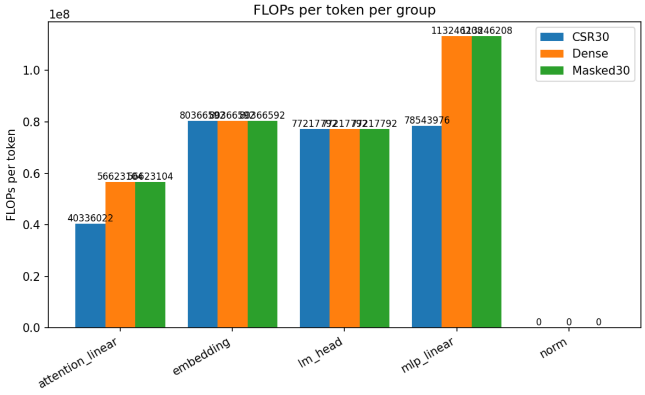


Figure 5: Per-token FLOPs proxy by group (Dense vs Masked30 vs CSR30). CSR reduces compute proxy in pruned linear groups.

6 Discussion

6.1 Interpreting Top-1 Agreement Critically

Top-1 agreement is intentionally minimal: it is deterministic, fast, and does not require generation benchmarks. However, because it is a strict argmax-based comparison, it should be read as a **behavioural proxy**, not a quality score. In particular, top-1 can overstate drift in ambiguous positions (small logit changes can flip argmax), and understate drift when confidence changes but argmax stays fixed. For this reason, the report always interprets top-1 together with structural metrics: Figure 3 explains *where* sparsity is injected, Figure 5 explains *what compute could change* under sparse execution, and Figure 2c reports *how local predictions react.*

6.2 Why Not Prune Everything? Choice vs Framework

Not pruning embeddings/`lm_head`/norms is primarily a methodological choice for stability and interpretability. Pruning frameworks allow broader pruning [3, 4], but pruning sensitive components complicates comparisons and can degrade behaviour disproportionately relative to their compute contribution. Here, restricting pruning to attention/MLP(feed-forward) linear maps isolates the effect on the main repeated linear operators (Figure 1) and keeps the analysis focused.

6.3 Hardware Constraints (What Is Measured vs What Is Not)

Limited RAM/VRAM prevented running larger models and large-scale evaluation. CSR inference benefits most from optimized sparse backends (often GPU-focused), which were not always available. Therefore, conclusions are explicitly limited to: (i) structural changes (storage, sparsity localization, FLOPs proxies) and (ii) logits-level behavioural similarity. Real latency and perplexity remain future work.

7 Conclusion

This report provides a controlled characterization of unstructured pruning in decoder-only Transformer LMs [1, 2] using a transparent Dense/Masked/CSR comparison. At 30% pruning on attention and MLP(feed-forward) linear maps, Masked and CSR show comparable top-1 agreement (Figure 2c), indicating that behavioural drift is driven by the pruning mask rather than by CSR storage. Structurally, CSR turns sparsity into physical savings: it reduces stored parameters (Figure 2a) and lowers a FLOPs proxy in pruned groups (Figure 5), consistent with the requirement that sparse execution must scale with non-zeros to yield compute savings [4, 7]. Overall, the framework links pruning location (Figure 3), structural impact, and behavioural stability in a way that remains reproducible under constrained hardware.

8 Future Work

The present study establishes a controlled baseline linking pruning location, structural effects, and behavioural drift under strict hardware constraints. Several natural extensions follow directly from these observations.

- **Exploring higher sparsity regimes.** Extending pruning beyond 30% (e.g., 50%, 70%, 90%) would allow us to characterize the *stability–sparsity curve* and identify potential tipping points where behavioural similarity collapses.

- **Beyond unstructured magnitude pruning.** While magnitude pruning provides a clean baseline, stronger one-shot methods such as SparseGPT or Wanda incorporate layer sensitivity and/or activation information. Comparing them within the same framework would separate limitations of the pruning *criterion* from limitations of sparsity itself [5, 6].

- **Accuracy recovery after pruning.** The measured behavioural drift is obtained *without* any recovery step. Light post-pruning fine-tuning, LoRA-based adaptation, or quantization-aware sparsification could recover similarity while retaining structural gains.

- **Structured and semi-structured sparsity.** Unstructured masks are harder to exploit efficiently on hardware. Evaluating N:M or block sparsity would better align the sparsity pattern with GPU-friendly sparse kernels and modern accelerators [4, 7].

- **From theoretical proxies to real acceleration.** This work relies on FLOPs-per-token proxies rather than wall-clock latency. Future experiments should measure real inference speedups using GPU-supported sparse backends (e.g., cuSPARSELt, Triton, PyTorch 2.x), across batch sizes and sequence lengths, to quantify the gap between theoretical and practical gains.

- **Scaling up and kernel-level benchmarking.** The current conclusions are based on small/medium models due to RAM/VRAM constraints. Applying the same pruning + analysis pipeline to larger checkpoints (when hardware permits) would test whether the observed behavioural/structural trends persist at scale and whether CSR-style storage yields larger end-to-end benefits. In parallel, an HPC-oriented extension is to *generate optimized implementations* of compute kernels (e.g., GEMM and related primitives such as Cholesky and FFT) using the dense model versus the sparsified model, then benchmark both generated codes under identical settings to assess whether sparsification degrades the *quality of code optimizations* produced by the model.

A From Logits to Token Predictions

This appendix provides a detailed explanation of the prediction pipeline used by decoder-only language models. It is included to make the definition of top-1 agreement fully explicit and self-contained.

Logits. For each token position (b, t), the model outputs a vector of logits $\ell_{bt} \in \mathbb{R}^V$, where V is the vocabulary size. Each logit is an unnormalized score associated with a candidate next token.

Softmax. To convert logits into a probability distribution, the softmax function is applied:

$$p_{bt}(k) = \frac{e^{\ell_{btk}}}{\sum_{j=1}^{V} e^{\ell_{btj}}}.$$

Softmax preserves the ordering of logits while normalizing them into probabilities.

Argmax decoding. Under greedy decoding, the predicted token is the index with the highest probability:

$$\hat{k}_{bt} = \arg\max_k \, p_{bt}(k).$$

Since softmax is strictly monotonic with respect to its inputs, this is equivalent to:

$$\hat{k}_{bt} = \arg\max_k \, \ell_{btk}.$$

Connection to top-1 agreement. Top-1 agreement therefore measures whether two models select the same most-likely token at each position when evaluated on identical inputs, independently of probability calibration or sampling strategy.

References

[1] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. Lin, I. M. Malik, A. Paszke, D. R. Radev, L. Stojanov, and A. Zettlemoyer, “OPT: Open Pre-trained Transformer Language Models,” *arXiv preprint arXiv:2205.01068*, 2022.

[2] S. Biderman, K. Black, Q. Gao, L. Golding, E. Hecht, F. Leahy, B. McDonell, J. Phang, N. Skowron, and S. Thite, “Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling,” *arXiv preprint arXiv:2304.01373*, 2023.

[3] S. Han, J. Pool, J. Tran, and W. J. Dally, “Learning both Weights and Connections for Efficient Neural Networks,” *Advances in Neural Information Processing Systems (NeurIPS)*, 2015.

[4] T. Gale, E. Elsen, and S. Hooker, “The State of Sparsity in Deep Neural Networks,” *arXiv preprint arXiv:1902.09574*, 2019.

[5] A. Frantar and G. Alistarh, “SparseGPT: Massive Language Models Can Be Accurately Pruned in One Shot,” *International Conference on Machine Learning (ICML)*, 2023.

[6] Y. Sun, Y. Zhao, T. Lin, Q. Wang, H. Zhou, and Z. Zhang, “A Simple and Effective Pruning Approach for Large Language Models,” *arXiv preprint arXiv:2306.11695*, 2023.

[7] NVIDIA, “NVIDIA Ampere Architecture Whitepaper,” 2020.