Compute Where It Counts: Adaptive Compute Allocation for Large Language Models via Learned Granular Sparsity

Anonymous ACL submission

Abstract

The inference of Large Language Models (LLMs) requires massive amounts of computation. Sparsity-aware inference pipelines can alleviate this cost by reducing the number of parameters used in each forward pass. We introduce “granular sparsity”, a novel method for reducing compute requirements. By decomposing matrix columns, the standard unit of sparsity, into smaller *stripes*, we create a flexible method of conditional computation that is more expressive than existing sparsity strategies.

Furthermore, we introduce a novel method for learning and controlling sparsity, which is inspired by sparse autoencoders. Notably, our method allows the model to designate different levels of sparsity to different input and layers. We validate our methods by distilling 2-6x more compute-efficient sparse language models from Llama 3.2 1B. Interestingly, we show evidence that our model allocates more computation to answering questions that humans deem more difficult.

1 Introduction

Large Language Models (LLMs) have revolutionized the field of machine learning, and have been scaled to unprecedented size and training costs. As a result, large compute budgets are now allocated for inference, in addition to training. To minimize these impacts, a great deal of recent research has been devoted to improving LLM inference efficiency. Approaches include Mixture of Experts (MoE) models, model distillation, quantization, and sparsification.

Most if not all of these techniques allocate a **uniform** number of active parameters or FLOPs (floating-point operations) towards ingesting and generating tokens. In contrast, we believe that certain tasks and sequences are fundamentally more or less “challenging” and should therefore receive compute budget allocations to match.

In this paper, we introduce the *granular sparsity operation* as a more expressive alternative to the commonly used column sparsity pattern (Lee et al., 2024). Furthermore, we demonstrate a method to train models that dynamically allocate different amounts of compute for different tokens and different layers. To do this, we directly learn sparsity thresholds and include a sparsity penalty in the loss function.

We convert and distill a Llama 3.2 1B Instruct model (Grattafiori et al., 2024) into “granular” models that consume 2x-6x fewer FLOPs per token. Examining the FLOPs assigned by the “granular” models to benchmark tasks reveals that these models naturally allocate less compute to “easier” tokens (such as role tokens, filler words, system prompt) and sequences (such as questions from ARC-Easy vs ARC-Challenge (Clark et al., 2018)).

2 Related Work

2.1 Activation Sparsity

Relufication (Mirzadeh et al., 2024) involves replacing pretrained LLM activation functions with ReLUs and inserting ReLUs elsewhere in the model. After finetuning to recover performance, Relufication can reduce FLOP counts by up to 50% with almost no degradation. Deja Vu (Liu et al., 2023) predicts sparsity on the fly by training small auxiliary MLPs. Q-Sparse (Wang et al., 2024) discards all but the top-K largest channels of input vectors when computing linear layers. They demonstrate better performance compared to a dense model with the same amount of compute, and also show that sparsity degrades performance less on larger models. Most similar to our work is CATS (Lee et al., 2024), which uses fixed thresholds to determine sparsity.

2.1.1 Mixture of Experts

The Mixture of Experts (MoE) conditional computing paradigm activates certain sections of the neural network ("experts") to reduce the active parameter count. Unlike activation sparsity methods, MoE architectures typically use a learned routing mechanism to choose which experts to activate. Sparsely-Gated Mixture-of-Experts (Shazeer et al., 2017) proposed a gating network that incentivizes sparse, yet balanced, expert selection for language modeling and machine translation. DeepSeekMoE (Dai et al., 2024) demonstrated that combinatorial expert selection with a larger number of experts leads to better performance. Expert Choice Routing (Zhou et al., 2022) demonstrates that higher performance can be achieved if different tokens can receive different amounts of compute. CompeteSMoE (Pham et al., 2024) demonstrates that choosing the experts with the largest output magnitude is an effective routing strategy.

2.2 Sparse Autoencoders

Sparse Autoencoders (SAEs) can faithfully reconstruct the hidden state of a neural network while activating a very small percentage of their features. Variants include top-k SAEs (Gao et al., 2024) that explicitly keep only the k largest activations, and JumpReLU (Rajamanoharan et al., 2024) SAEs that pass activations features through a ReLU activation and keep only the activations that exceed a learned threshold. Notably, JumpReLU makes sparsity learnable, and allows different numbers of features to activate for different examples. Other work (Ayonrinde, 2024) has also shown that reconstruction fidelity is improved when different numbers of features can be activated for different tokens.

3 Methods

3.1 Granular Sparsity

Existing sparsity methods (Mirzadeh et al., 2024; Wang et al., 2024; Lee et al., 2024) exploit column sparsity (sometimes transposed and described as row sparsity) in matrix multiplications: when an input vector has an element that is exactly zero, the computation for the corresponding matrix column can be skipped. Here, a column is either entirely used or entirely unused. Drawing from the insight of DeepSeekMoE (Dai et al., 2024) that more conditional computation combinations leads to better performance, we create a more expressive sparsity mechanism by breaking each column into a set of *stripes* - with each stripe being activated individually. This greatly increases the number of achievable sparsity configurations (see Figure 1).

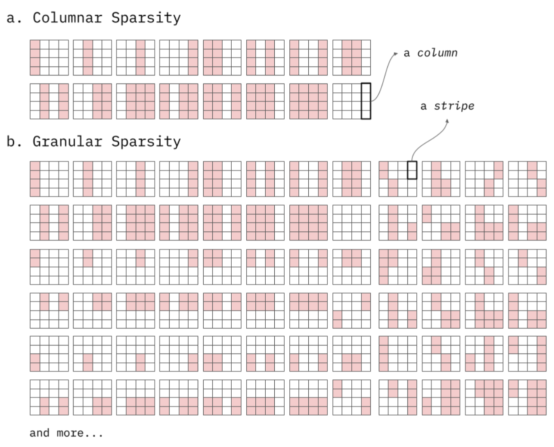


Figure 1: Given a 4x4 matrix, using column sparsity results in 16 configurations. Combining contiguous sets of two rows into stripes where each stripe can be independently controlled results in significantly more configurations.

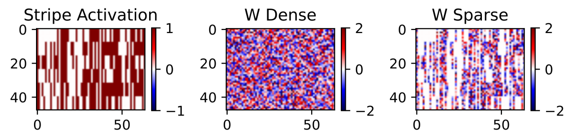


Figure 2: For each input X we multiply by a masked version of the weights. The mask is created by checking if a channel of x is between the thresholds and thus a zeroed region. These thresholds are parameterized along each input channel and repeated in groups (stripes) along each output channel.
Repetition along output channels is important because it allows us to avoid multiple multiply adds for only one mask check for the whole region.

Formally, consider the multiplication of a matrix $W \in \mathbb{R}^{m\times n}$ with the vector $x \in \mathbb{R}^n$, resulting in the vector $y \in \mathbb{R}^m$.

$$y := Wx$$

This operation can be broken into the sum of the column vectors of W (with v_i denoting the i-th column of W), weighted by the elements x_i of x:

$$y = \sum_{i=1}^{n} x_i v_i$$

Standard column-wise sparse matrix multiplication, which we denote as $\text{SMM}(x, W, \mathcal{M})$, uses a mask $\mathcal{M} \in \{0,1\}^n$ to zero out some elements of x, so that only the remaining elements are required for the matrix multiplication. We use the set $S_{\mathcal{M}} = \{1 \le i \le n \mid \mathcal{M}_i = 1\}$ to denote the set of indices where the elements of x are not zero.

$$y_{\text{sparse}} = \text{SMM}(x, W, \mathcal{M}) := \sum_{i=1}^{n} \mathcal{M}_i x_i v_i$$

$$= \sum_{i \in S_{\mathcal{M}}} x_i v_i$$

In our method, we partition the vector y into k equally sized *stripes*, such that $y = [y^{(1)}, y^{(2)}, ..., y^{(k)}]$. We similarly partition the column vectors of v_i of W such that $v_i = [v_i^{(1)}, v_i^{(2)}, ..., v_i^{(k)}]$. With some abuse of notation, we use Ξ as a concatenation analogue of the summation operator $\sum$, such that $y = \Xi_{r=1}^{k} y^{(r)}$. This means that the original matrix multiplication can be written as:

$$y = \Xi_{r=1}^{k} \sum_{i=1}^{n} x_i v_i^{(r)}$$

Our granular sparsity operation, which we denote as $\text{GMM}(x, W, \mathcal{G})$, uses a mask $\mathcal{G} \in \{0,1\}^{k \times n}$ that zeroes out some of the stripes in W, as shown in Figure 2. We define $S_{\mathcal{G}} = \{(r,i) : 1 \le r \le k,\ 1 \le i \le n \mid \mathcal{G}_{r,i} = 1\}$ to be the set of stripe indices that are not zeroed out.

$$y_{\text{granular}} = \text{GMM}(x, W, \mathcal{G}) := \Xi_{r=1}^{k} \sum_{i=1}^{n} \mathcal{G}_{r,i} x_i v_i^{(r)}$$

$$= \Xi_{r=1}^{k} \sum_{(r,i) \in S_{\mathcal{G}}} x_i v_i^{(r)}$$

Note that when $k = 1$, granular matrix multiplication is the same as standard column sparsity.

3.2 Sparsity Thresholds

We use a different mask $\mathcal{G}$ for each input vector x (a strategy known as *contextual sparsity*) (Liu et al., 2023). To determine the $\mathcal{G}$, we use the magnitudes of each element in x. Specifically, we learn a grid of thesholds $\theta \in \mathbb{R}_{+}^{k \times n}$ such that $\mathcal{G}_{r,i}$ is 1 if and only if x_i has have a magnitude of at least $\theta_{r,i}$. We define this relation using the Heaviside step function $H(z)$.

$$H(z) := \begin{cases} 1 & z \ge 0 \\ 0 & z < 0 \end{cases}$$

$$\mathcal{G}_{r,i} = H(|x_i| - \theta_{r,i})$$

We denote granular matrix multiplication parameterized by thresholds with $\text{GMM}(x, W, \theta)$:

$$y_{\text{granular}} = \text{GMM}(x, W; \theta)$$

$$:= \Xi_{r=1}^{k} \sum_{i=1}^{n} H(|x_i| - \theta_{r,i})\ x_i v_i^{(r)}$$

We initialize θ to zero at the start of training. Furthermore, to prevent θ from becoming negative, we set $\theta = \max(0, \theta)$ after every parameter update. Finally, we found that θ benefited from a significantly higher learning rate than other parameters. We set the learning rate of θ equal to the base learning rate multiplied by $\eta_\theta \sqrt{n}$, where η_θ is a hyperparameter.

3.3 Learning Thresholds

Previous works that use thresholds to determine contextual sparsity masks have often relied on heuristics to determine the threshold values (Lee et al., 2024). Instead, we seek better optimization be directly learning the thresholds. Unfortunately, $H(z)$ is not differentiable. We therefore build on the ideas introduced in JumpReLU (Rajamanoharan et al., 2024) to construct a straight-through-estimator (Bengio et al., 2013) with a pseudo-derivative that approximates the true derivative. This pseudo-derivative is defined as follows, with K representing a kernel function and ϵ representing a tunable bandwidth:

$$\frac{\partial}{\partial z} H(z) := \frac{1}{\epsilon} K\left(\frac{z}{\epsilon}\right)$$

For our kernel function K, we use the rectangle function seen in JumpReLU:

$$K(z) := H\left(z - \frac{1}{2}\right) - H\left(z + \frac{1}{2}\right)$$

When calculating either $\frac{\partial}{\partial x_i}\mathcal{G}_{r,i}$ or $\frac{\partial}{\partial \theta_{r,i}}\mathcal{G}_{r,i}$ we set the corresponding ϵ_i equal to the batch-wise standard deviation of x_i (which does not receive gradients), scaled by a constant uniform hyperparameter α_ϵ.

$$\epsilon_i := \alpha_\epsilon \text{std}(x_i)$$

For a more detailed analysis of this gradient estimator, we refer readers to the JumpReLU paper (Rajamanoharan et al., 2024).

3.4 Normalization

When initializing from a pre-trained network, we found that the batch-wise scales and offsets of x_i

values can vary throughout the network, making it difficult to tune hyperparameters. To remedy this, we whiten x before it enters the matrix multiplication operation. Concretely, for a batch of x vectors, we calculate the batch-wise mean $\bar{x} \in \mathbb{R}^n$ and standard deviation $\sigma(x) \in \mathbb{R}^n$. Then, we then perform the following:

$$y_{\text{granular}} = \text{GMM}\Big(x - \bar{x}, W; \theta \odot \sigma(x)\Big) + W\bar{x}$$

Note that when θ is composed of zeroes, this whitening procedure does not effect the value of y_{sparse}. To increase training stability, we track $\bar{x}$ and $\sigma(x)$ on a rolling basis. For this, the hyperparameter β_{dist} is used to compute an exponential moving average. At inference time the running $\bar{x}$ and $\sigma(x)$ values from the last step of training are used.

3.5 Straight-Through Estimation

Previous work (Wang et al., 2024) shows that sparse models can benefit from using straight-through estimators during training. In our case, that means taking the gradients of x as if there was no sparsity. Specifically, we use the following definition for the gradients of x:

$$\nabla_x \, y_{\text{granular}} := W^\top$$

Although this modification obviously leads to biased gradients, we theorize that this estimator improves performance by removing the variance imparted on the grads when the values of $\mathcal{G}$ are changing frequently. The STE could also fix the vanishing gradients associated with high sparsity, as postulated by Q-Sparse (Wang et al., 2024).

The gradients of θ and W are left unchanged.

3.6 Controlling Sparsity

A key advantage of the learned threshold scheme is that we can control the sparsity of the model using a loss function. For this, we first calculate the number of floating point operations (FLOPs) to compute $\text{GMM}(x, W; \theta)$, which is given by:

$$\text{FLOPs}(x, W, \mathcal{G}) := \frac{m}{k}||\mathcal{G}||_1$$

We then define $\text{FLOPs}(B)$ to represent the number of operations required to operate on a batch B, $\text{FLOPs}_{\text{base}}(B)$ to represent the number of operations required if we did not have sparsity, and $\text{FLOPs}_{\text{target}}(B)$ to represent the desired FLOP count. Finally, we define the Flop Reduction Ratio (FRR) to be the ratio between the base FLOP count and the sparse FLOP count:

$$\text{FRR}(B) := \frac{\text{FLOPs}_{\text{base}}(B)}{\text{FLOPs}(B)}$$
$$\text{FRR}_{\text{target}}(B) := \frac{\text{FLOPs}_{\text{base}}(B)}{\text{FLOPs}_{\text{target}}(B)}$$

Our loss is then the following:

$$\mathcal{L}_{\text{FLOPs}} = \Big[\min\Big(\text{FRR}(B) - \text{FRR}_{\text{target}}(B), 0\Big)\Big]^2$$

This loss was chosen because it gives us precise control over the desired compute costs, and gives us stable performance during training. Importantly, we found it important to include a warmup phase at the start of training where $\text{FLOPs}_{\text{target}}(B)$ is incrementally lowered. For this, we linearly increase $\text{FRR}_{\text{target}}(B)$ from a starting value, usually 1.5, to our final target value.

We also found it necessary to regulate the FLOP costs of the language modeling head. If left unchecked, the sparsity of stripes corresponding to infrequent tokens would approach zero. Therefore, we detach the gradients leading to the FLOP calculation of the head once its FRR reaches a set value, usually 10.

3.7 Feed-Forward Network Modifications

Previous works have found that the intermediate activations of the feed-forward blocks exhibit natural sparsity (Mirzadeh et al., 2024). To leverage this, we slightly modify our granular sparsity system for use in the feed-forward blocks. The Llama 3 suite of models uses gated linear units (GLU) (Shazeer, 2020) as their feedforward design. These are parameterized by a gate matrix $W_{\text{gate}} \in \mathbb{R}^{n\times d}$, an up matrix $W_{\text{up}} \in \mathbb{R}^{n\times d}$, and a down matrix $W_{\text{down}} \in \mathbb{R}^{d\times n}$:

$$a = \text{silu}(W_{\text{gate}}x)$$
$$y_{\text{GLU}} = W_{\text{down}}\Big(W_{\text{up}} \odot a\Big)$$

In our method, we compute $W_{\text{gate}}x$ with the standard granular sparsity method. Then, we use our learned threshold method to compute $\mathcal{M} \in \{0,1\}^d$ based on the magnitudes of $\text{silu}(W_{\text{gate}}x)$. We use this mask as follows:

$$a = \mathcal{M} \odot \text{silu}\big(\text{GMM}(x, W_{\text{gate}}, \mathcal{G})\big)$$
$$y_{\text{GLU, granular}} = W_{\text{down}}\Big(W_{\text{up}} \odot a\Big)$$

When the operations rendered unnecessary by the mask are filtered out, the FLOP cost of this operation is then:

$$\text{FLOPs}(\mathcal{G}, \mathcal{M}) = \frac{d}{k}||\mathcal{G}||_1 + 2n||\mathcal{M}||_1$$

3.8 Model Distillation

To efficiently train our sparse models, we use knowledge distillation (Hinton et al., 2015) with a teacher network (usually the same model that was used to initialize the sparse model's parameters). For our distillation loss, we use a combination of the forward KL divergence (FKL) and the reverse KL divergence (RKL), which has been shown to work better than either divergence individually (Wu et al., 2024). For a sequence of length T, this gives us:

$$\text{FKL} := \sum_{t=1}^{T} \text{KL}\Big(p_{\text{teacher}}(y_t|y_{<t}), p_{\text{student}}(y_t|y_{<t})\Big)$$

$$\text{RKL} := \sum_{t=1}^{T} \text{KL}\Big(p_{\text{student}}(y_t|y_{<t}), p_{\text{teacher}}(y_t|y_{<t})\Big)$$

$$\mathcal{L}_{\text{distill}} := \frac{1}{2}\Big(\text{FKL} + \text{RKL}\Big)$$

Our total loss is a weighted combination of our distillation loss and our FLOP loss described in subsection 3.6:

$$\mathcal{L} = \mathcal{L}_{\text{distill}} + \lambda_{\text{FLOPs}}\mathcal{L}_{\text{FLOPs}}$$

4 Experiments

4.1 Setup

We tested our methods with the Llama-3.2-1B-Instruct [1] model. This model (henceforth referred to as the "base" model) was used for both the teacher and the initialization of the student. We used the AdamW optimizer. Training was conducted over approximately 800M tokens, taking 48 hours on a cluster of 8 H100 GPUs (for a total of 384 GPU hours). Details of our training data can be found in Appendix A and detailed hyperparameters can be found in Appendix B.

4.2 Ablation Studies

To determine the impact of design decisions made in this work, we perform four ablation experiments. First, we set the number of stripes to 1 (alternatively, this ablation can be thought of as standard column sparsity). Second, we remove the gradual warmup to our target FLOPs ratio (this means the model is trained to obtain an 6x FLOP reduction on the very first training step). Third, we remove the normalization described above. In the fourth ablation, we remove the limit on FRR for the language modeling head. The performance of ablated models is discussed in subsection 4.3.

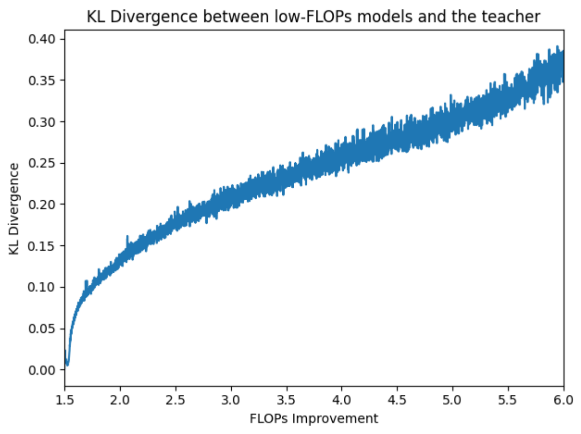


Figure 3: As the FRR target of the training run is increased, the FKL between the thresholded model's logits and the teacher logits increase.

4.3 Results

The performance of thresholded models with increasing Flop Reduction Ratios (FRRs) on MMLU (Hendrycks et al., 2021), ARC (Clark et al., 2018), HellaSwag (Zellers et al., 2019), WinoGrande (Sakaguchi et al., 2019), BoolQ (Clark et al., 2019) and SciQ (Welbl et al., 2017) is shown in Table 1. Table 2 compares the performance of thresholded models with state-of-the-art (fully activated) transformer models of similar size. All results are reported from a single training and evaluation run.

Collectively, we observe a steady decrease in performance across most benchmarks as the FRR is increased. This is explained by the increasing KL divergence between high FRR models and the original model (Figure 3). Note that the stability of performance on WinoGrande and BoolQ is because the base model's performance is close to random choice.

The 2x and 6x FRR models perform similarly to small transformer models in their compute classes that have been trained from scratch (namely SmolLM-360M[2] and SmolLM-360M[3]). We note that the FRR models outperform SmolLM models

[1]unsloth/Llama-3.2-1B-Instruct

[2]HuggingFaceTB/SmolLM-135M
[3]HuggingFaceTB/SmolLM-360M

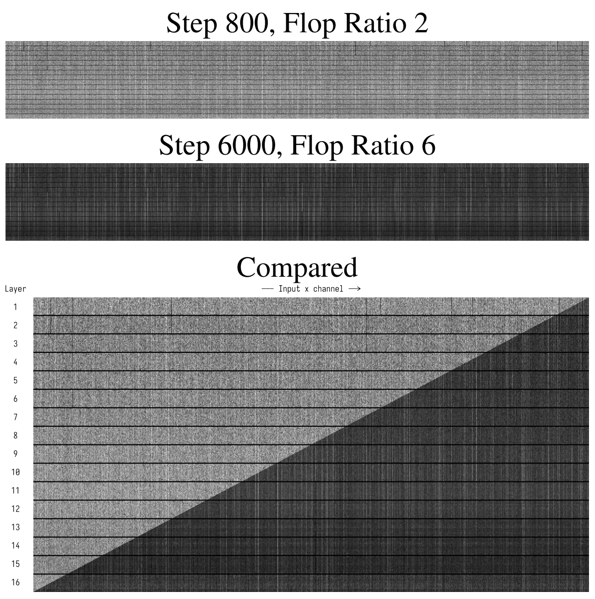


Figure 4: Activation frequencies of W_{gate} at 2x and 6x FRR. The W_{gate}s across layers are stacked vertically. The rows of the matrices correspond to weight stripes. The intensity of each column represents the frequency that a input position passes the threshold for the column (a darker column indicates lower frequency). The "Compared" chart overlays the "Step 800" and "Step 6000" charts to demonstrate how important features (brighter columns) emerge early in training and are magnified in relative importance over the training run.

on MMLU, match on WinoGrande and underperform on ARC/HellaSwag. We believe these differences arise from differences in training data.

Table 3 presents the performance of ablated models (described in subsection 4.2) on MMLU and ARC. Removing the FRR warm-up and restrictions on the FRR of the language modeling head results in immediate model collapse and performance equivalent to random choice.

5 Discussion

5.1 Sparsity Patterns

Analyzing the activation frequencies across matrices and layers of the model reveals several interesting patterns about the structure of circuits and sparsity within our models.

First, we observe particularly important channels in the residual stream. These are channels that activate for almost every input across many layers. They emerge very early (in as few as 100 training steps), and stay almost unchanged for the entirety of training. Figure 4 demonstrates this for the W_{gate} matrix. We believe that these channels may capture common knowledge, and serve a similar purpose to the shared experts used by DeepSeekMoE (Dai et al., 2024).

Next, we notice that among the Q, K, and V attention matrices, activation of the V matrix is the most dense, followed by K and O (Figure 5). Note the dip in activation frequencies across matrices from layers 8-14 followed by a slight rise in layers 15 and 16.

We find striking patterns in the activation frequencies of the O attention matrix (Figure 6). Often, individual attention heads will have consistently high or low activations across channels. We hypothesize that the network could be learning to implicitly "prune" unhelpful attention heads, similar to previous work that reduces compute cost by explicitly removing attention heads (Mugnaini et al., 2025).

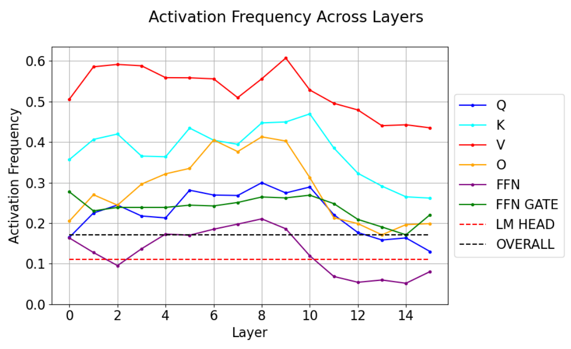


Figure 5: Activation frequency of different matrix types across layers for a 6x FRR model. Note that FFN represents the W_{up} and W_{down} matrices, while FFN GATE represents the W_{gate} matrix.

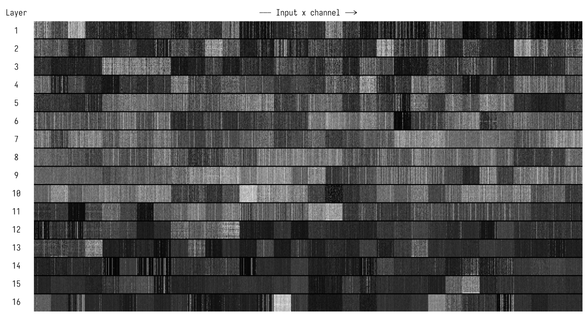


Figure 6: The sparsity levels *within* O heads tend to be similar across channels. Sometimes entire heads will be always on or always off.
Note that unlike the QKV, UP, GATE, there are no patterns across layers as the input to O is not the residual stream.

As referenced earlier, we found an important pattern in the sparsity of the language modeling head:

FRR	Average	MMLU	ARC-C	ARC-E	Hellaswag	WinoGrande	BoolQ	SciQ
1x	57.1	48.3	54.8	72.4	41.1	52.5	53.3	77.6
2x	54.5	43.6	47.5	66.8	38.8	51.9	60.1	72.6
3x	51.5	40.3	44.1	59.9	34.1	49.5	60.1	72.3
4x	48.4	37.9	38.6	57.5	33.4	50.3	55.3	65.5
5x	46.4	35.7	37.3	55.6	30.7	50.3	54.8	60.7
6x	44.0	33.8	32.1	49.8	29.4	53.0	53.6	56.4

Table 1: Model performance across benchmarks at different Flop Reduction Ratios. Note that the 1x FRR model is simply the original model's (`unsloth/Llama-3.2-1B-Instruct`) performance.

Model Type	MMLU	ARC	Hellaswag	WinoGrande
3x FRR (330M params)	40.3	44.1	34.1	49.5
SmolLM-360M	34.17	51.1	53.8	53.7
6x FRR (160M params)	33.76	32.08	29.39	53.02
SmolLM-135M	30.23	43.99	42.3	52.7

Table 2: Comparison of 3x and 6x FRR models and compute-equivalent small models. SmolLM performance is self-reported.

stripes corresponding to earlier vocabulary tokens have higher activation frequencies. This is because the vocabulary of the Llama model family is implicitly sorted from high to low token frequency, so frequent tokens have more compute designated to them than infrequent tokens. This effect is so pronounced that without regularization, the network will eventually put nearly *zero* compute towards infrequent tokens, leading to representation collapse.

5.2 Variable Compute Budgets

Our use of sparsity thresholds means that different tokens and different sequences can use different amounts of compute.

Consider the FLOPs spent by the 6x FRR model on processing questions from the ARC-Easy vs ARC-Challenge dataset in Figure 7. The average FLOPs dedicated by the 6x FRR model to processing tokens of questions in the ARC-Easy and ARC-Challenge benchmarks follow similar distributions but with the Arc-Easy offset to the left indicating that a significant number of the ARC-Easy questions were allocated less compute than for ARC-Challenge questions. As evidenced by the scores of the 6x FRR model on these benchmarks (Table 1), the ARC-Easy questions are indeed easier for the 6x FRR model!

We can also visualize the compute spent on individual tokens of a question-answer sequence. Figure 8 demonstrates three common trends in compute allocation that we observe in general:

- Punctuation and filler words such as "How-

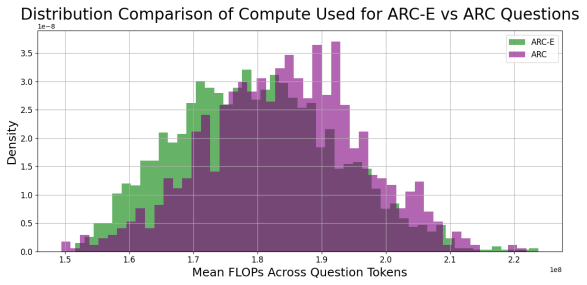
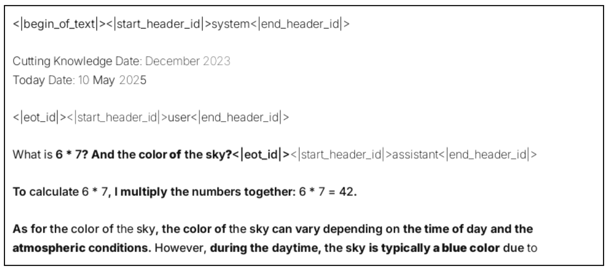


Figure 7: We calculate the FLOPs allocated to each token in all of the questions of ARC-Easy and ARC-Challenge. We ignore the system prompt and only consider the tokens in the question and four options. These values are averaged across tokens in a question and each question is represented as a sample in the histogram above. The model dedicates more compute to questions from the ARC Challenge dataset compares to the ARC Easy dataset.



```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

Cutting Knowledge Date: December 2023
Today Date: 10 May 2025

<|eot_id|><|start_header_id|>user<|end_header_id|>

What is 6 * 7? And the color of the sky?<|eot_id|><|start_header_id|>assistant<|end_header_id|>

To calculate 6 * 7, I multiply the numbers together: 6 * 7 = 42.

As for the color of the sky, the color of the sky can vary depending on the time of day and the
atmospheric conditions. However, during the daytime, the sky is typically a blue color due to
```

Figure 8: Active parameter count allocation across different token positions in a prompt and generation pair. The thickness of the font indicates the number of active parameters (and thus the compute budgets) allocated to different tokens (thicker text indicates more compute). The text following the assistant role token was generated by a 6x FRR model.

ARC Scores (%)					
FRR	**Baseline**	**No Warmup**	**No LMHead Limit**	**No Normalization**	**No Stripes**
2x	47.5	22.4	29.8	41.3	31.2
3x	44.1	22.4	26.0	42.4	25.7
4x	38.5	22.4	25.7	37.3	24.7
5x	37.2	22.4	25.7	33.6	23.5
6x	32.1	22.4	25.7	29.1	21.2
MMLU Scores (%)					
FRR	**Baseline**	**No Warmup**	**No LMHead Limit**	**No Normalization**	**No Stripes**
2x	48.3	22.7	29.5	32.5	30.1
3x	43.6	22.6	24.9	29.8	24.9
4x	40.3	22.6	24.1	29.1	24.7
5x	35.7	21.0	24.1	26.2	25.6
6x	33.8	21.0	24.1	28.2	22.5

Table 3: ARC and MMLU Performance under different ablation settings across various FRR levels.

ever" are low budget.

- Behavior on system prompt and system/user/assistant role tokens gets distilled into very few active parameters and thus receives a very low compute allocation.
- Semantically quoting sections of user prompt such as "6 * 7" and "the color of the sky" uses fewer active parameters.

5.3 Effects of Hyperparameters

We found that performance was stable with respect to the bandwidth scale α_ϵ within a range of about 0.05 to 0.25. However, values outside of this range caused significant training instability.

Furthermore, the choice momentum parameter of running batch statistics β_{dist} had very little impact on performance, with values tested in the range of 0.9 to 0.99.

When calculating pseudo-derivatives, we found that other kernel functions besides the rectangle kernel described in subsection 3.3 gave similar performance.

6 Future Work

Achieving a theoretical FLOPs reduction is not directly linked with practical CPU/GPU clock time reductions. For large stripe sizes which are multiples of the SIMD size, acceleration of inference of CPU is straightforward (one can first compute the mask, then use it to skip multiply adds), but for GPU it remains less tractable.

Our striping method groups output channels based on their order. However, outside of attention heads, there is no guarantee that adjacent channels are functionally similar. Therefore, when initializing a sparse model from a pretrained one, it may be beneficial to reorder channels in order to form semantic groupings. This idea has already seen success in the realm of mixture-of-expert conversions (Elazar and Taylor, 2022).

From Figure 3 we note that the KL divergence between the base model and FRR models remains stable between 2x and 4.5x FRR. It then rises above 4.5x FRR. We believe performance improvements could be realized by modifying the FRR warmup schedule to something other than the linear one used here.

7 Ethical Consideration & Potential Risks

Since the patterns learned by granular sparsity are dependent on training data distributions, during inference the model could under-allocate compute to tokens that are important but were underrepresented during training). This may lead to lower robustness or bias amplification for marginalized dialects or identity groups. In addition, any efficiency gains resulting from this technique could larger-scale deployment of LLM inference for malicious applications.

Limitations

The primary limitation of our work concerns the scale of our experiments. We did not train models larger than 1B parameters, and we did not train for longer than 1B tokens. We believe that significantly better benchmark performance could be achieved with more training. Previous work (Wang et al., 2024) has also indicated that sparsity leads to less performance degradation at larger scales, so our method be be more suitable for larger models than those tested here.

Furthermore, we only tested our method on transformer architectures and language modeling. The application to other models, such as such as vision transformers (Dosovitskiy et al., 2021) has not been explored.

References

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, and 3 others. 2025. Smollm2: When smol goes big – data-centric training of a small language model. *Preprint*, arXiv:2502.02737.

Guoqing Zheng Shweti Mahajan Dany Rouhana Andres Codas Yadong Lu Wei-ge Chen Olga Vrousgos Corby Rosset Fillipe Silva Hamed Khanpour Yash Lara Ahmed Awadallah Arindam Mitra, Luciano Del Corro. 2024. Agentinstruct: Toward generative teaching with agentic flows. *Preprint*, arXiv:2407.03502.

Kola Ayonrinde. 2024. Adaptive sparse allocation with mutual choice feature choice sparse autoencoders. *Preprint*, arXiv:2411.02124.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. Boolq: Exploring the surprising difficulty of natural yes/no questions. *Preprint*, arXiv:1905.10044.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *Preprint*, arXiv:1803.05457.

Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. 2024. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *Preprint*, arXiv:2401.06066.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. *Preprint*, arXiv:2010.11929.

Nathan Elazar and Kerry Taylor. 2022. Implicit mixture of interpretable experts for global and local interpretability. *Preprint*, arXiv:2212.00471.

Leo Gao, Tom Dupré la Tour, Henk Tillman, Gabriel Goh, Rajan Troll, Alec Radford, Ilya Sutskever, Jan Leike, and Jeffrey Wu. 2024. Scaling and evaluating sparse autoencoders. *Preprint*, arXiv:2406.04093.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. *Preprint*, arXiv:2009.03300.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *Preprint*, arXiv:1503.02531.

Donghyun Lee, Je-Yong Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. 2024. Cats: Contextually-aware thresholding for sparsity in large language models. *Preprint*, arXiv:2404.08763.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, and Beidi Chen. 2023. Deja vu: Contextual sparsity for efficient llms at inference time. *Preprint*, arXiv:2310.17157.

Seyed Iman Mirzadeh, Keivan Alizadeh-Vahid, Sachin Mehta, Carlo C del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. 2024. ReLU strikes back: Exploiting activation sparsity in large language models. In *The Twelfth International Conference on Learning Representations*.

Leandro Giusti Mugnaini, Bruno Lopes Yamamoto, Lucas Lauton de Alcantara, Victor Zacarias, Edson Bollis, Lucas Pellicer, Anna Helena Reali Costa, and Artur Jordao. 2025. Efficient llms with amp: Attention heads and mlp pruning. *Preprint*, arXiv:2504.21174.

Quang Pham, Giang Do, Huy Nguyen, TrungTin Nguyen, Chenghao Liu, Mina Sartipi, Binh T. Nguyen, Savitha Ramasamy, Xiaoli Li, Steven Hoi, and Nhat Ho. 2024. Competesmoe – effective training of sparse mixture of experts via competition. *Preprint*, arXiv:2402.02526.

Senthooran Rajamanoharan, Tom Lieberum, Nicolas Sonnerat, Arthur Conmy, Vikrant Varma, János

Kramár, and Neel Nanda. 2024. Jumping ahead: Improving reconstruction fidelity with jumprelu sparse autoencoders. *Preprint*, arXiv:2407.14435.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. Winogrande: An adversarial winograd schema challenge at scale. *Preprint*, arXiv:1907.10641.

Noam Shazeer. 2020. Glu variants improve transformer. *Preprint*, arXiv:2002.05202.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *Preprint*, arXiv:1701.06538.

Teknium. 2023. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants.

Hongyu Wang, Shuming Ma, Ruiping Wang, and Furu Wei. 2024. Q-sparse: All large language models can be fully sparsely-activated. *Preprint*, arXiv:2407.10969.

Johannes Welbl, Nelson F. Liu, and Matt Gardner. 2017. Crowdsourcing multiple choice science questions. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, pages 94–106, Copenhagen, Denmark. Association for Computational Linguistics.

Taiqiang Wu, Chaofan Tao, Jiahao Wang, Runming Yang, Zhe Zhao, and Ngai Wong. 2024. Rethinking kullback-leibler divergence in knowledge distillation for large language models. *Preprint*, arXiv:2404.02657.

Weizhe Yuan, Jane Yu, Song Jiang, Karthik Padthe, Yang Li, Dong Wang, Ilia Kulikov, Kyunghyun Cho, Yuandong Tian, Jason E Weston, and Xian Li. 2025. Naturalreasoning: Reasoning in the wild with 2.8m challenging questions. *Preprint*, arXiv:2502.13124.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? *Preprint*, arXiv:1905.07830.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Tianle Li, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zhuohan Li, Zi Lin, Eric. P Xing, Joseph E. Gonzalez, Ion Stoica, and Hao Zhang. 2023. Lmsys-chat-1m: A large-scale real-world llm conversation dataset. *Preprint*, arXiv:2309.11998.

Yanqi Zhou, Tao Lei, Hanxiao Liu, Nan Du, Yanping Huang, Vincent Zhao, Andrew Dai, Zhifeng Chen, Quoc Le, and James Laudon. 2022. Mixture-of-experts with expert choice routing. *Preprint*, arXiv:2202.09368.

Setting	**Value**
Base model	Llama-3.2-1B-Instruct
Max sequence length	1024
FRR target	6
FLOPs warmup length	6000 steps
Learning rate schedule	7000 steps
Optimizer	Adam
Beta1	0.9
Beta2	0.95
Weight Decay	0.01
β_{dist}	0.99
α_{ϵ}	0.1
η_{θ}	10.0
λ_{FLOPs}	1.0

Table 4: Hyperparameter settings for the default model

A Hyperparameters

The hyperparameters for our training run are presented in Table 4. Note that the default training mode had a FLOP warmup, had normalization enabled and placed a limit on the FLOPs improvement allowed for the LM Head. These settings were turned off, one at a time, to conduct ablation settings (see subsection 4.2).

B Training Data

The training data used in our distillation process is listed below:

- OpenHermes-2.5 Training Data[4] (Teknium, 2023)
- NaturalReasoning Dataset[5] (Yuan et al., 2025)
- SmolTalk Dataset[6] (Allal et al., 2025)
- Orca AgentInstruct-1M-v1[7] (Arindam Mitra, 2024)
- LMSYS-Chat-1M Dataset[8] (Zheng et al., 2023)
- MMLU training split (repeated 5×) (Hendrycks et al., 2021)
- ARC training split (repeated 5×) (Clark et al., 2018)
- WinoGrande training split (repeated 5×) (Sakaguchi et al., 2019)

All data sequences were converted to the standard chat format used by Llama-3.2-1B-Instruct, then filtered for a maximum total sequence length

[4]teknium/OpenHermes-2.5
[5]facebook/natural_reasoning
[6]HuggingFaceTB/smoltalk
[7]microsoft/orca-agentinstruct-1M-v1
[8]lmsys/lmsys-chat-1m

of 1024. We also packed shorter sequences together to increase training efficiency, and used attention masking to prevent interactions between packed sequences.