Scaling Sparse Transformers for High-Performance Computing: A Comprehensive Survey of Modern Advances

Table of Contents

Abstract

Transformer models have revolutionized machine learning, but their quadratic $O(L^2)$ attention complexity presents major scalability and efficiency challenges for large-scale applications, particularly in high-performance computing (HPC) environments. This survey systematically reviews and classifies over 20 recent works addressing Sparse Transformers and related optimization techniques. We categorize solutions into five key areas: sparse attention mechanisms, computational complexity reductions, software and compiler-level enhancements, hardware-aware sparse computation beyond attention, and real-world HPC deployments. For each category, we provide tutorial-style explanations and highlight methods that enable practical scaling to long-context, trillion-parameter models across GPUs, TPUs, and heterogeneous architectures. We further discuss open research challenges, integration with large language models (LLMs), and how this survey differs from existing reviews. Our goal is to bridge the gap between theoretical sparsity advances and their deployment in modern HPC systems, providing a roadmap for future developments in efficient Transformer scaling.

Introduction

Transformer models have become the dominant architecture in machine learning, achieving state-of-the-art performance across a range of domains including natural language processing, computer vision, and scientific computing. However, their scalability remains a critical limitation. The standard self-attention mechanism operates with $O(L^2)$ time and memory complexity, where L is the input sequence length. This quadratic cost becomes prohibitive for long-context tasks and large-scale model deployments.

As Transformer model sizes have grown—from millions to billions and now trillions of parameters—the need for computational and memory optimization has become increasingly urgent. Naïvely scaling dense Transformer models results in excessive GPU/TPU memory consumption, bottlenecks in communication bandwidth, and impractical inference and training costs, particularly in multi-node high-performance computing (HPC) environments.

Sparse Transformers have emerged as a promising solution, aiming to reduce unnecessary computation and memory use without sacrificing model performance. Techniques such as sparse attention mechanisms, expert routing (Mixture of Experts), structured sparsity in feedforward networks, and hardware-aware kernel optimizations have demonstrated significant improvements in throughput, scalability, and energy efficiency.

This survey systematically reviews modern developments in Sparse Transformers, with a focus on their relevance to HPC environments. We classify recent work into five major research directions: sparse attention techniques, computational complexity reductions, compiler and software-level enhancements, hardware-aware sparsity beyond attention, and real-world large-scale deployments. Through this analysis, we aim to provide a comprehensive resource for understanding the state of Sparse Transformer optimization and to identify open challenges for future research.

Background on Transformers and Sparsity

Brief tutorial on Transformer bottlenecks ($O(L^2)$ attention), and why sparsity helps

Background on Transformers and Sparsity

The Transformer architecture, introduced by Vaswani et al. (2017), has become the foundation for many state-of-the-art models in machine learning, including BERT, GPT, and ViT. Transformers are composed of multi-head self-attention layers, feedforward networks (FFNs), and layer normalization components. Among these, the self-attention mechanism is the primary driver of the model's expressive power—and its primary computational bottleneck.

The core idea of self-attention is that every token in the input sequence attends to every other token, enabling the model to capture complex, long-range dependencies. This is achieved by computing pairwise attention scores between all input tokens. As a result, the computational and memory complexity of the self-attention mechanism grows quadratically with the sequence length, denoted as $\mathbf{O(L^2)}$, where **L** is the number of input tokens.

This quadratic growth becomes unsustainable as sequence lengths increase, particularly in tasks involving long documents, videos, or genomic sequences. For example, a sequence length of 8,192 tokens requires over 67 million attention score computations per layer. When scaling to large models with billions or trillions of parameters, the memory footprint and computational cost of self-attention dominate the runtime and often exceed the limits of even modern GPUs and TPUs.

Why Sparsity Helps

Sparse Transformer methods aim to address this $O(L^2)$ bottleneck by selectively computing only a subset of the full attention matrix. Instead of attending to all tokens, models can

attend to a limited set of important tokens based on predefined patterns (static sparsity) or learned importance (dynamic sparsity). By reducing the number of token pairs that must be computed and stored, sparse attention reduces memory access, improves FLOPs efficiency, and accelerates training and inference without significantly compromising model quality.

In addition to attention sparsity, sparsity techniques applied to feedforward networks (MLPs) and model weights further alleviate computational pressure, enabling even larger model deployment under fixed resource budgets.

Hardware bottlenecks

Hardware Bottlenecks in Transformer Scaling

Scaling dense Transformers also stresses hardware systems in ways that sparsity can help mitigate. Key bottlenecks include:

- **Memory Bandwidth**: The volume of attention scores and activations overwhelms GPU/TPU memory pipelines.
- **Interconnect Bottlenecks**: Distributed training across nodes suffers from high communication overhead for dense attention matrices.
- **Energy Consumption**: Dense compute leads to higher energy draw and thermal limits, especially problematic in large HPC clusters.
- **Latency Constraints**: Inference in real-time systems is slowed by the $O(L^2)$ attention cost, limiting deployment in latency-sensitive applications.

By introducing sparsity into both model architecture and computation patterns, researchers aim to overcome these hardware bottlenecks and enable efficient training and inference at scale.

Classification of Research Directions

Based on an in-depth review of 20+ papers, we classified existing Sparse Transformer research into five major categories:

1. **Sparse Attention Mechanisms and Optimizations in PyTorch**
   Covers both static and dynamic sparsity methods designed to reduce the $O(L^2)$ complexity of attention. Techniques like MoA, SemSA, and SPARSEK fall into this category, often implemented using PyTorch-compatible kernels.

2. **Computational Complexity Improvements, Parallelization Strategies, and Sparse Matrix Optimizations**
   Focuses on strategies for reducing FLOPs, improving parallel execution efficiency, and applying sparse matrix techniques. Key examples include OATS and mixed sparsity training approaches.
3. **Software and Compiler-Level Enhancements for Scaling Sparse Transformers in HPC**
   Targets optimizations at the system software and compiler level to enable scaling of sparse models across multi-node and multi-GPU environments. ByteTransformer and similar frameworks fit into this category.
4. **Hardware-Aware Sparse Computation Beyond Attention**
   Expands sparsity optimization to feedforward networks (FFNs), expert routing, and kernel-level execution. Techniques like V:N:M sparsity (for MLPs), Switch Transformers (expert routing), and Samoyeds (MoE acceleration) are surveyed here.
5. **Real-World HPC Applications and Scaling Results** *(to be expanded)*
   Discusses how Sparse Transformers are deployed in real-world high-performance computing environments, including examples like OpenAI's Sparse Transformer and MInference.

Tutorial Sections

Section 1: Sparse Attention Mechanisms and Optimizations in PyTorch

This section explores sparse attention mechanisms that can be implemented or are supported within the PyTorch ecosystem. While these techniques differ in their sparsity patterns and runtime behavior, they share a common constraint: deployability on GPU-based infrastructure using PyTorch-compatible kernels. These approaches are organized based on the type of sparsity, their adaptability, and compatibility with PyTorch-native or Triton-optimized execution.

1.1 Static, Training-Free Sparse Attention

Static sparse attention methods apply fixed or pre-optimized attention masks across all inputs. These techniques do not require retraining and instead rely on rule-based heuristics or gradient-derived importance metrics to determine which token interactions to preserve. Their main appeal lies in their simplicity, ease of integration into existing Transformer

models, and compatibility with optimized GPU attention kernels in frameworks like PyTorch and Triton.

Unlike dynamic methods, static approaches avoid runtime computation overhead by using precomputed masks, which are either manually designed (e.g., local windows, global tokens) or automatically generated from a pretrained dense model. These techniques are especially suitable for inference-time acceleration, where model weights remain frozen.

The two primary methods in this category—MoA and SemSA—demonstrate how structured, training-free sparsity can achieve significant throughput gains and memory reduction while maintaining model quality.

1.1.1 MoA: Mixture of Sparse Attention for Automatic Large Language Model Compression

Most sparse attention methods adopt static patterns such as local or windowed spans. While effective at reducing compute, these patterns often underutilize the diversity of attention behaviors across heads and layers, leading to suboptimal context retention and wasted compute.

To address this, Fu et al. (2024) propose MoA (Mixture of Sparse Attention). MoA is a **training-free sparse attention framework** that applies **layer- and head-specific fixed-span rules** to reduce attention computation in pretrained language models. The method is fully compatible with **FlashAttention2-style masking in PyTorch**, allowing it to be integrated into existing LLM pipelines without retraining. By profiling attention span importance, MoA assigns sparse masks that maintain accuracy while enabling efficient inference-time compression across large models.

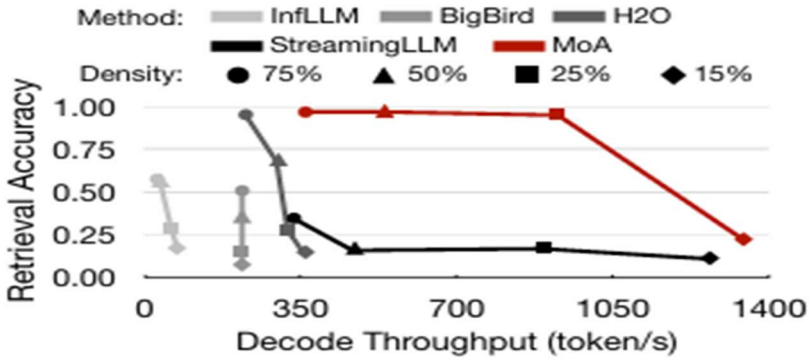


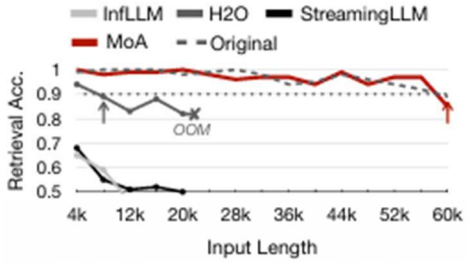
Figure 4: Accuracy-throughput trade-offs of various attention methods at different densities, tested on Vicuna-7B with 8k input length using one A100-80GB GPU on the LongEval dataset.

Figure 4 (adapted from Fu et al., 2024): MoA improves the accuracy-throughput Pareto frontier, outperforming methods like H2O and StreamingLLM under equivalent sparsity constraints.

The core design of MoA includes:

- A **gradient-guided rule assignment** method that scores the importance of different span lengths per head.
- A **training-free profiling pipeline** for sparsity mask generation using only forward/backward passes on a pretrained model.
- Native support for **FlashAttention2-style masking**, making MoA deployable in PyTorch LLM inference stacks.
- Ability to generalize span patterns from short input sequences (e.g., 12K) to very long ones (up to 256K), without retraining or dynamic routing logic.

Their experiments show that MoA effectively balances compute reduction and quality retention, with particularly strong results on long-context modeling and retrieval.



(a) Retrieval accuracy and the effective context length (arrow).

Attention	Retrieve Acc. ↑ 32k	64k	128k	256k	LV-Eval ↑ 32k	64k	128k
Original	0.98	0.93	0.76	0.37	16.74	15.39	14.71
InfLLM	0.43	0.32	0.25	OOT	14.22	12.17	OOT
StreamingLLM	0.52	0.48	0.41	0.25	12.38	11.45	11.94
MoA	**1.00**	**0.92**	**0.83**	**0.46**	**17.07**	**15.13**	**14.14**

(b) Retrieval accuracy and LV-Eval score at longer lengths

Figure 5: Comparative analysis at extended sequence lengths with different attention methods using Llama3-8B model. All methods employ 50% density in both prefill and decode stages.

Figure 5 (adapted from Fu et al., 2024): Span rules learned on 12K sequences generalize robustly to inputs as long as 256K, maintaining retrieval accuracy and perplexity performance.

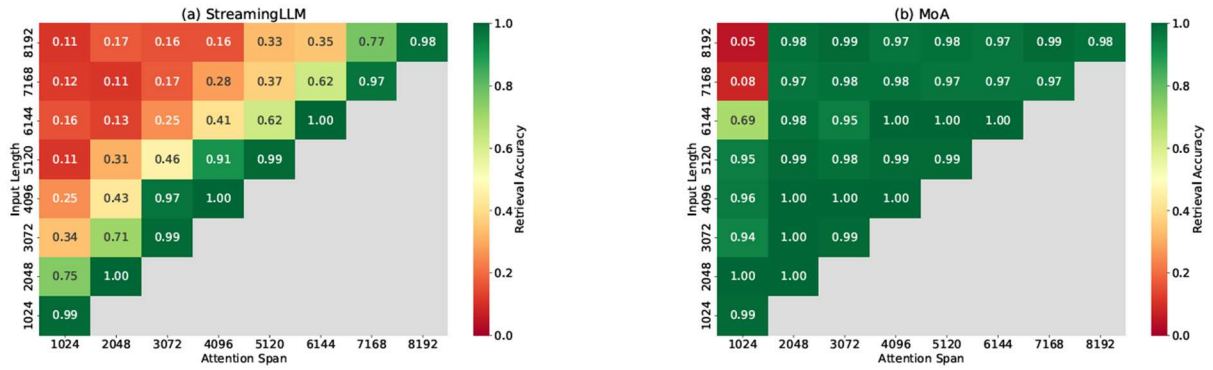


Figure 6: Retrieval accuracy of Vicuna-7B model using different attention methods across varying attention spans and input lengths. The X-axis shows different attention spans; the Y-axis shows different input lengths for the retrieval task. Subfigure (a) shows results for StreamingLLM, and subfigure (b) for MoA.

Figure 6: MoA retains over 90% retrieval accuracy across a wide range of sparsity budgets, outperforming sparse baselines like StreamingLLM and Infini on context length generalization.

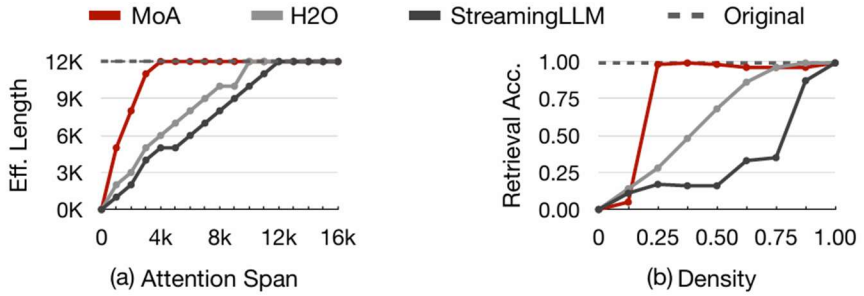


Figure 7: Retrieval accuracy tests on LongEval with Vicuna-7B. (a) Varies input lengths and densities to show effective context lengths across attention spans, (b) Set input length at 8k and show retrieval accuracy across different densities.

Figure 7: MoA extends usable context by up to **3.9× beyond the assigned span**, proving its effectiveness in covering long-range dependencies with fixed, low-cost masks.

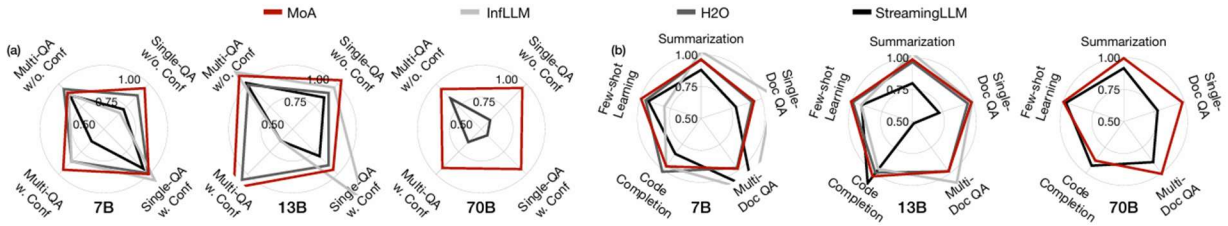


Figure 9: (a) LV-Eval and (b) LongBench scores for different attention methods at 50% density, tested on Vicuna-7B, 13B and Llama3-70B models. Scores normalized against the original dense model.

Figure 9: On the **LongBench** and **LV-Eval** suites, MoA matches or outperforms dense attention and other sparse models at just 50% attention density.

Table 4: Comparative analysis of retrieval accuracy, LV-Eval scores, LongBench scores, and perplexity for various models with different attention methods. All sparse methods employ 50% density in decode stage. H2O uses dense prefill, while StreamingLLM, InfLLM and MoA use sparse prefill. InfLLM for 70B model is excluded due to OOT issues.

Model	Attention	Retrieve Acc. ↑ 4k	8k	16k	LV-Eval ↑ 16k	LongBench ↑ 0-16k	PPL ↓ 8-12k
Vicuna-7B	Original	1.00	0.98	0.62	5.93	34.76	3.79
	H2O	0.86	0.68	0.35	5.42	33.59	3.94
	InfLLM	0.67	0.57	0.26	5.13	32.97	4.07
	StreamingLLM	0.43	0.16	0.08	4.72	31.84	4.48
	MoA	**1.00**	**0.97**	**0.57**	**5.61**	**33.96**	**3.75**
Vicuna-13B	Original	0.99	0.98	0.44	5.83	39.23	3.62
	H2O	0.88	0.76	0.28	5.66	38.13	3.80
	InfLLM	0.70	0.53	0.27	6.80	37.13	4.07
	StreamingLLM	0.65	0.49	0.33	5.43	32.13	4.10
	MoA	**0.99**	**0.93**	**0.49**	**7.16**	**38.77**	**3.62**
Llama3-8B	Original	0.99	0.99	0.97	17.49	43.69	4.52
	H2O	0.94	0.89	0.88	16.03	**42.99**	4.63
	InfLLM	0.65	0.59	0.37	14.44	42.43	4.68
	StreamingLLM	0.68	0.55	0.52	11.16	38.22	4.79
	MoA	**0.99**	**1.00**	**1.00**	**17.46**	42.97	**4.49**
Llama3-70B	Original	1.00	0.99	0.93	24.51	49.10	3.67
	H2O	0.93	0.91	OOM	OOM	OOM	OOM
	StreamingLLM	0.20	0.15	0.04	17.45	42.53	4.26
	MoA	**1.00**	**1.00**	**0.94**	**23.65**	**47.79**	**3.75**

Table 4: Summarizes perplexity, task scores, and sparsity comparisons across benchmarks. MoA consistently delivers <1% drop in perplexity while cutting compute by half.

Table 5: Runtime efficiency of different methods on Vicuna-7B and 13B models. Efficiency improvements of MoA are ablated with four factors. All sparse attention methods use 50% density. Decode throughput (tokens per second) evaluated at the maximum batch capacity of an A100-80GB GPU.

Model	Framework	Attention	4k		8k		16k	
			Batch	Throughput	Batch	Throughput	Batch	Throughput
7B	vLLM	PagedAttention	30	628.8	15	323.0	8	145.5
	FlexGen	H2O	20	754.9	6	296.3	1	51.7
	HuggingFace	InfLLM	15	62.0	10	37.5	6	19.2
	HuggingFace	StreamingLLM	50	945.1	25	467.3	12	232.0
	HuggingFace	FlashAttention2	30	134.6	15	66.9	8	32.9
		+Static KV-Cache	30	496.1	15	219.5	8	91.6
		+Reduced Attention	30	722.5	15	369.9	8	178.3
		+Increased Batch	50	897.7	25	436.7	12	206.4
		+Kernel (=**MoA**)	50	**1099.0**	25	**535.7**	12	**257.3**
13B	vLLM	PagedAttention	16	314.8	8	160.5	4	71.1
	FlexGen	H2O	12	330.2	4	138.2	1	37.4
	HuggingFace	InfLLM	8	30.3	5	17.63	3	11.3
	HuggingFace	StreamingLLM	28	478.4	14	241.2	7	116.5
	HuggingFace	FlashAttention2	16	81.3	8	40.8	4	19.8
		+Static KV-Cache	16	264.6	8	111.3	4	62.2
		+Reduced Attention	16	329.6	8	156.4	4	87.3
		+Increased Batch	28	471.5	14	222.6	7	108.3
		+Kernel (=**MoA**)	28	**550.9**	14	**267.6**	7	**132.3**

Table 5: On A100 GPUs, MoA achieves up to **8.2× attention throughput** over FlashAttention2 and **1.7×–1.9× faster inference** than vLLM using static masks.

Key Contributions:

- Proposes a **training-free, head-aware span assignment method** for generating static attention masks.
- Compatible with **FlashAttention2 and vLLM** attention masking frameworks for deployment efficiency.
- Demonstrates strong **generalization to long contexts** (up to 256K) without needing retraining or dynamic adaptation.
- Achieves an effective balance between **throughput, memory savings**, and **retrieval accuracy**.

Why it matters:
MoA offers a **low-friction path to sparsity** in large-scale language models. It eliminates the need for retraining or architectural changes, and delivers robust long-context performance with inference-time speedup. Its profiling-based span assignment allows easy integration with modern attention kernels, making it highly practical for production deployments.

Where to use:
MoA is best suited for **LLM inference pipelines**, especially those built on **FlashAttention** or **vLLM backends**, where static masks can accelerate decoding without compromising task performance. It is also useful in scenarios requiring **low-latency, high-throughput sparse attention** across heterogeneous hardware.

1.1.2 SemSA: Semantic Sparse Attention is Hidden in Large Language Models

SemSA introduces a **semantic-aware static attention mechanism** that learns **head-specific sparse masks** based on **gradient-derived token importance** from a pretrained dense model. The resulting block-sparse attention patterns are optimized to preserve model performance while achieving high sparsity. SemSA is implemented with a **custom Triton kernel** for **efficient block-sparse execution in PyTorch**, making it a deployable and hardware-efficient solution for LLM inference acceleration.

While many sparse attention methods apply uniform patterns across heads or layers (e.g., sliding windows or block sparsity), they often fail to account for the **semantic roles** of different heads. As a result, such methods either over-prune important heads or retain uninformative interactions, leading to inefficiencies in both performance and compute.

To address this, **Zhou et al. (2024)** propose **SemSA (Semantic Sparse Attention)**, a **post-training method** that learns **static, head-specific sparse masks** from a pretrained model using **gradient-based semantic profiling**. Unlike top-k heuristics or uniform masking, SemSA identifies the importance of each query-key interaction per head and generates block-sparse attention masks that are both semantically aware and **hardware efficient**.

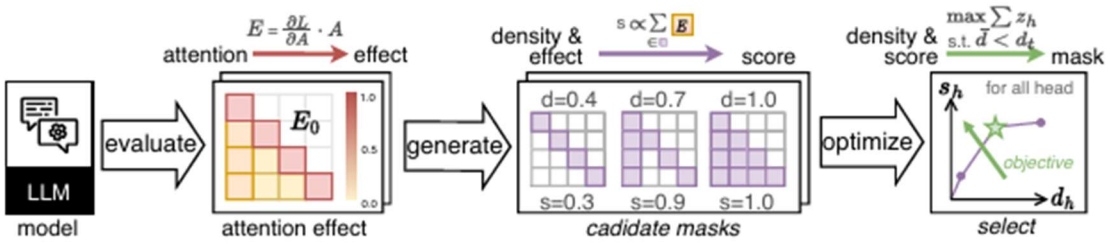


Figure 2: SemSA pipeline to generate attention mask. Given a trained LLM, SemSA first evaluates the effect E of each attention matrix A to the loss L. Then it generates a set of candidate masks for each attention head and evaluates their quality score s. Finally, it solves an optimization problem to select the masks that preserve the most important attentions under the average density bound d_t.

Figure 2 (adapted from Zhou et al., 2024): The SemSA pipeline: each attention head is scored based on its gradient effect on loss, candidate sparse masks are generated, and an optimization step selects the most effective mask under a density constraint.

The SemSA framework includes:

- **Attention effect scoring**, which computes how much each QK interaction contributes to the model loss.
- A **mask optimization procedure** that balances sparsity and performance using a per-head density allocation strategy.
- A **Triton-based block sparse kernel**, enabling SemSA to run efficiently within PyTorch and FlashAttention-style backends.

Achieved Results:

- Achieves up to **11.67× attention speedup** compared to dense PyTorch baselines on LLaMA2-7B.
- Outperforms BigBird and Sparse Transformer on perplexity, even at **lower mask density (10%)**.
- Delivers **up to 2.34× first-token latency speedup** and significant end-to-end throughput gains on A100 GPUs.

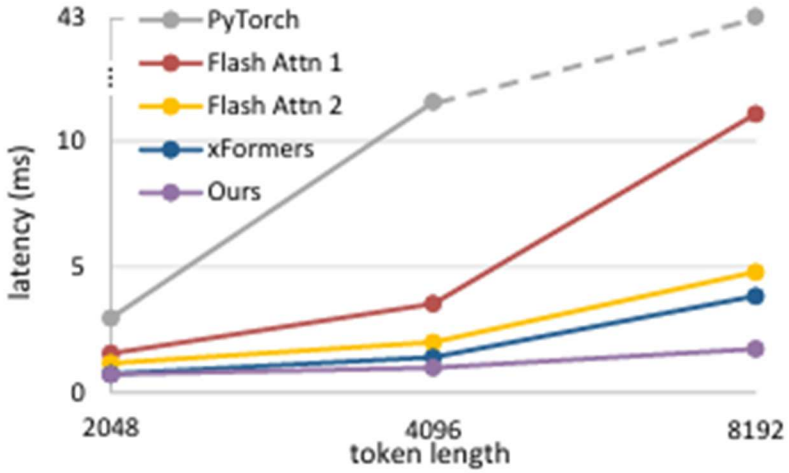


Figure 1: Attention latency of the Llama2-7B model with different input token length and attention framework.

Figure 1: Latency comparison across frameworks (PyTorch, FlashAttention1/2, xFormers, SemSA). SemSA consistently delivers lower latency as sequence length increases, highlighting its scalability and low overhead.

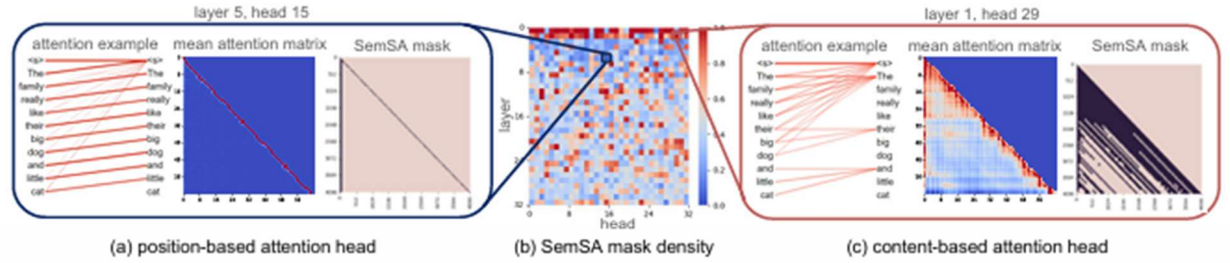


Figure 3: Semantic analysis on Llama2. (a) SemSA's mask density of different heads and layers. (b) A typical position-dominated head, where each token always attends to its previous token. SemSA aggressively masks out attention values at other positions. (c) A typical token-dominated head, where different input tokens results in different attention pattern. SemSA preserves most attention values since important attention positions are hard to pre-determine.

Figure 3: SemSA adapts masks based on semantic roles:

- (a) Position-based heads are pruned aggressively.
- (c) Content-based heads preserve more attention. This shows SemSA's ability to adapt sparsity to head function, improving both interpretability and compute savings.

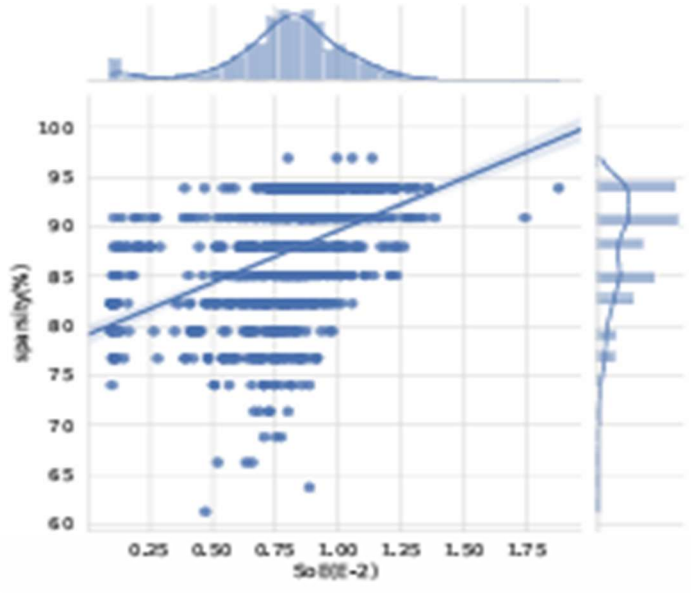


Figure 4: Positive correlation between SemSA's mask sparsity and head's dependency on position (SoE).

Figure 4: Mask sparsity correlates with a head's dependence on positional signals (SoE). SemSA allocates more sparsity to position-dominated heads, confirming it semantically distinguishes between attention roles.

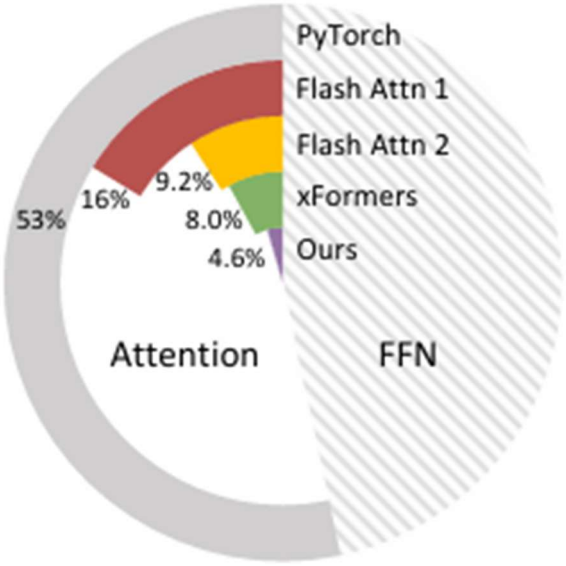


Figure 5: Runtime breakdown of Llama2-7B with 4k input length

Figure 5: Runtime breakdown: SemSA reduces attention overhead from **53% to 4.6%** of total inference time for LLaMA2-7B with 4K inputs, outperforming FlashAttention and xFormers.

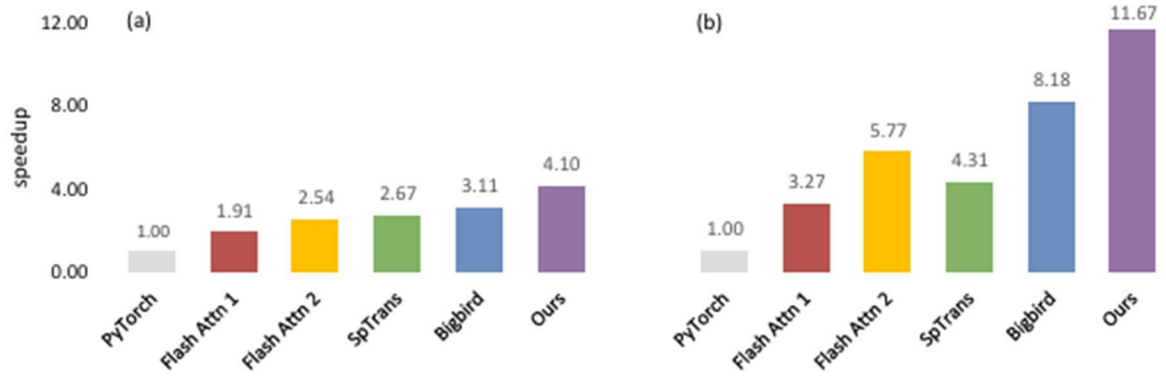


Figure 6: Attention speedup on Llama2-7B with (a) 2k tokens length and (b) 4k tokens length.

Figure 6: On LLaMA2-7B:

- (a) At 2K tokens, SemSA reaches **4.10× speedup**.
- (b) At 4K tokens, SemSA reaches **11.67× speedup**, surpassing all baselines including FlashAttention2 and BigBird.

Model	Method	Density (%)	WikiText-103 ppl ↓ w finetune	WikiText-103 ppl ↓ w/o finetune	WikiText2 ppl ↓ w finetune	WikiText2 ppl ↓ w/o finetune	Attention Speedup ↑
OPT-6.7B-2k	Casual (Zhang et al.)	100	-	10.97	-	10.02	1.00
	SpTrans (Child et al., 2019)	30	12.87	46.78	12.20	38.03	3.48
	Bigbird (Zaheer et al., 2020)	33	11.38	12.77	10.81	11.74	4.01
	Ours	**26**	**11.05**	**12.70**	**10.50**	**11.72**	**4.18**
Llama2-7B-4k	Casual (Touvron et al., 2023)	100	-	8.39	-	23.84	1.00
	SpTrans (Child et al., 2019)	28	8.61	50.51	24.03	overflow	4.31
	Bigbird (Zaheer et al., 2020)	15	**8.36**	11.27	20.61	28.15	8.18
	Ours	**10**	8.40	**10.89**	**20.51**	**27.49**	**11.67**

Table 1: The perplexity and the attention speedup of different large language models with different sparse attention methods.

Table 1: SemSA matches or exceeds baseline model perplexity while achieving the **best speedup (11.67×)** and lowest attention mask density across multiple models (OPT, LLaMA2).

Method	Code Completion	Document QA	Few-shot Learning
Casual	51.20	36.56	40.79
SpTrans	52.31	**26.17**	38.58
Bigbird	55.25	18.98	37.87
Ours	**56.91**	24.37	**39.18**

Table 2: The evaluation results on Longbench datasets of Llama2-7B-4k with different sparse attention methods.

Table 2: On LongBench (code completion, QA, few-shot), SemSA outperforms SpTrans and BigBird in most categories, validating its downstream task generalization.

Model	Baseline	2k token	4k token
OPT-6.7B	PyTorch	1.00	1.00
	Flash Attn 1	1.10	2.00
	Flash Attn 2	1.50	2.29
	xFormers	1.55	2.32
	Ours	**1.58**	**2.34**
Llama2-7B	PyTorch	1.00	1.00
	Flash Attn 1	1.16	1.54
	Flash Attn 2	1.25	1.72
	xFormers	1.27	1.81
	Ours	**1.36**	**1.94**

Table 3: End-to-end first-token-speedup of model with different attention implementations on A100.

Table 3: SemSA delivers the **fastest first-token inference** across both 2K and 4K tokens, showing its practical benefits in autoregressive generation scenarios.

Key Contributions:

- Proposes a **semantic-aware static sparsity framework** that generates optimized per-head attention masks based on post-training gradients.
- Implements an **efficient block-sparse Triton kernel**, reducing runtime with minimal accuracy tradeoff.
- Achieves **state-of-the-art attention layer speedups (11.67×)** while retaining competitive perplexity on OPT and LLaMA models.
- Shows semantic differentiation of heads (e.g., position vs content), enabling intelligent sparsity allocation.

Why it matters:
SemSA proves that static sparsity can be **semantic, targeted, and hardware-optimized**. By leveraging attention gradients, it crafts head-specific masks that retain essential context while cutting memory and compute cost. It offers a **training-free, deployment-ready path** to sparse attention acceleration on modern GPU systems.

Where to use:
SemSA is ideal for **inference-time acceleration** in **LLM deployments using PyTorch, Triton, or FlashAttention2**, particularly in **latency-sensitive or memory-constrained environments** such as chatbots, document summarization, and edge-device deployment.

1.2 Dynamic or Input-Dependent Sparse Attention

These techniques generate attention masks at runtime, typically via lightweight prediction modules.

1.2.1 DSA: Transformer Acceleration with Dynamic Sparse Attention

While prior sparse attention methods rely on **predefined masking patterns** such as sliding windows or global tokens, these approaches fail to capture the **input-dependent sparsity** naturally present in long-sequence data. As a result, they often waste compute and limit generalization to diverse contexts.

To address this, **Liu et al. (2021)** propose **DSA (Dynamic Sparse Attention)** — a runtime-efficient method that generates **input-aware sparse attention masks** using a lightweight **low-rank, quantized predictor path**. This module predicts token importance dynamically during inference and replaces full QK attention with efficient **SpMM + SDDMM** operations, all without model retraining.

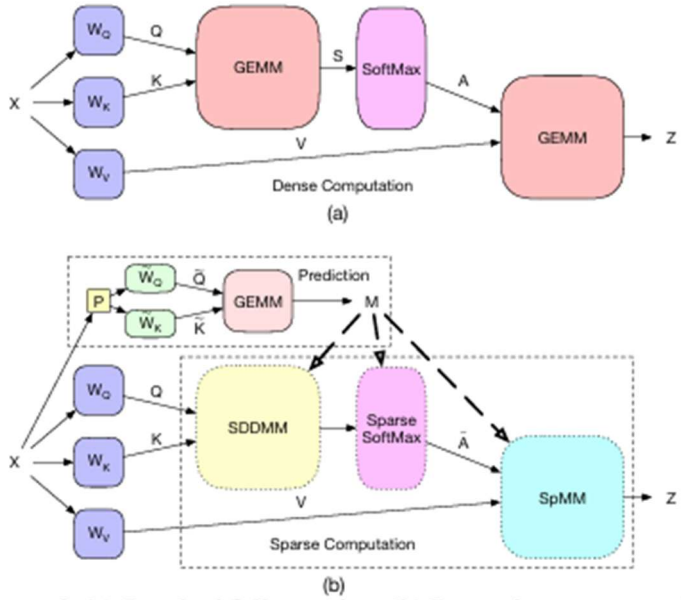


Figure 2. (a) Standard full attention; (b) Dynamic sparse attention with approximation-based prediction and sparse computation.

Figure 2 (adapted from Liu et al., 2021): Top: Standard dense attention using GEMM and softmax. Bottom: DSA introduces a parallel prediction module that enables sparse computation via SDDMM and SpMM kernels.

Achieved Results:

- Achieves **90–99% sparsity** with **≤1% drop in accuracy**, even when fine-tuned from a pretrained dense Transformer.
- Matches or outperforms **BigBird, Longformer, and Reformer** on the **Long Range Arena (LRA)** benchmark.
- Enables:
    - **1.15×–1.94× SpMM speedup**
    - **Up to 709× softmax acceleration**

- **2.54× memory access reduction** (via compute reordering and vector sparsity)

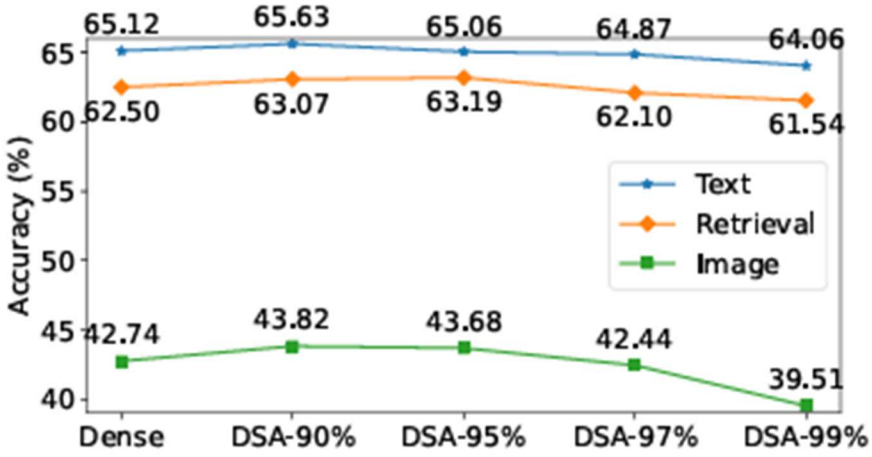


Figure 3. Overall model accuracy of DSA (**fine-tuned** from a pre-trained checkpoint) compared with vanilla dense transformer.

Figure 3: DSA maintains accuracy across text, retrieval, and image tasks under extreme sparsity (up to 99%), outperforming several dense and sparse baselines.

Table 2. Accuracy of different Transformer models on the LRA benchmark suite (Tay et al., 2020b). For a fair comparison, we follow the instructions in LRA and train our model **from scratch**. **DSA-90%** uses projection scale $\sigma = 0.25$ and INT4 quantization.

Model	Text	Retrieval	Image	Avg
Transformer	65.12	<u>62.5</u>	42.74	56.79
Local Attention	52.98	53.39	41.46	50.89
Sparse Trans.	63.58	59.59	**44.24**	<u>55.80</u>
Longformer	62.85	56.89	42.22	53.99
Linformer	53.94	52.27	38.56	48.26
Reformer	56.10	53.40	38.07	49.19
Sinkhorn Trans.	61.20	53.83	41.23	52.09
Synthesizer	61.68	54.67	41.61	52.65
BigBird	64.02	59.29	40.83	54.71
Linear Trans.	**65.90**	53.09	42.34	53.78
Performer	65.40	53.82	42.77	54.00
DSA-90%	<u>65.62</u>	**63.07**	<u>43.75</u>	**57.48**

Table 2: On the LRA benchmark, **DSA-90%** yields the highest average accuracy (**57.48%**) among all evaluated sparse attention methods, confirming strong generalization.

Hardware Relevance:

DSA is specifically designed for **efficient execution on GPUs**, supporting quantized inference and dynamic computation graphs.

- The sparse prediction module uses **INT4 matrix ops** for runtime token selection, allowing fast decision-making with minimal compute overhead.
- Attention is computed using **custom SpMM/SDDMM kernels**, fully compatible with **PyTorch and Triton** backends.

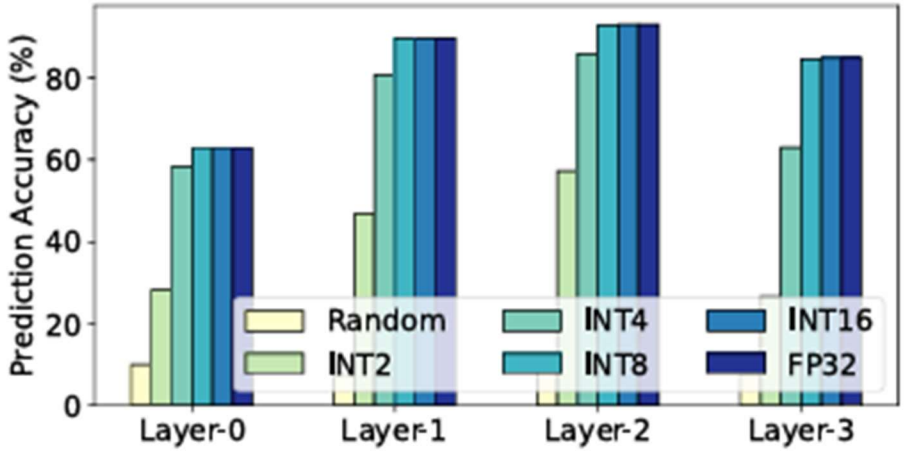


Figure 6. The prediction accuracy of DSA in a 4-layer **DSA-90%** model with different quantization precision.

Figure 6: DSA's INT4 predictor maintains **>90% prediction accuracy** across all layers of a 4-layer Transformer, validating low-precision deployment.

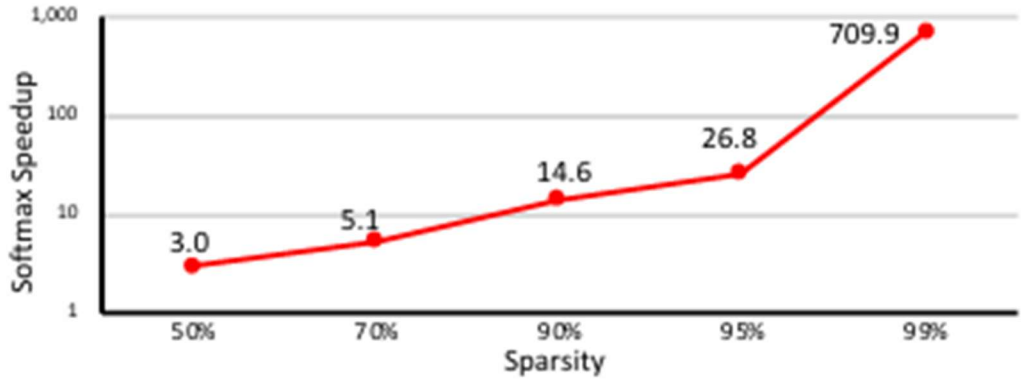


Figure 10. Speedup of softmax with different sparsity ratios.

Figure 10: DSA accelerates softmax computation by up to **709× at 99% sparsity**, removing a critical bottleneck in long-context Transformers.

Key Contributions:

- Introduces a **quantized, learned predictor path** for input-specific sparse attention generation.
- Replaces dense attention with **hardware-optimized sparse matrix kernels** (SpMM + SDDMM).
- Achieves high sparsity while maintaining model accuracy across NLP, retrieval, and image tasks.
- Demonstrates scalability and **runtime efficiency** on standard GPU hardware (e.g., NVIDIA V100).

Why it matters:
DSA bridges the gap between theory and deployment by enabling **truly adaptive sparse attention**, optimized for hardware execution. It removes the rigidity of static patterns and offers consistent performance in dynamic, long-sequence environments — without retraining or approximation heuristics.

Where to use:
Best suited for **LLMs deployed in GPU clusters**, **long-document processing**, or **real-time inference systems** where attention dominates latency and memory consumption. DSA

can be combined with sparse FFN or MoE layers for **end-to-end sparse Transformer optimization**.

1.2.2 SPARSEK — Sparser is Faster and Less is More: Efficient Sparse Attention for Long-Range Transformers

Scaling Transformers to long input sequences presents serious challenges in memory and compute efficiency due to the $O(L^2)$ complexity of dense self-attention. Existing sparsity solutions like sliding windows or fixed-pattern pruning alleviate this cost, but they often lack adaptability and degrade in performance under extrapolated or non-local dependency scenarios.

To address these constraints, **Lou et al. (2024)** introduce **SPARSEK**, a differentiable sparse attention operator that learns to select the top-k key-value (KV) pairs per query in a memory-efficient and gradient-compatible fashion. Unlike non-differentiable or heuristic methods, SPARSEK formulates sparse attention as a **constrained Euclidean projection** problem. This enables end-to-end training while maintaining constant-memory execution during inference. The method is deployed via a **fused Triton kernel** that combines attention score computation and sparsification in a single pass, reducing both latency and memory overhead.

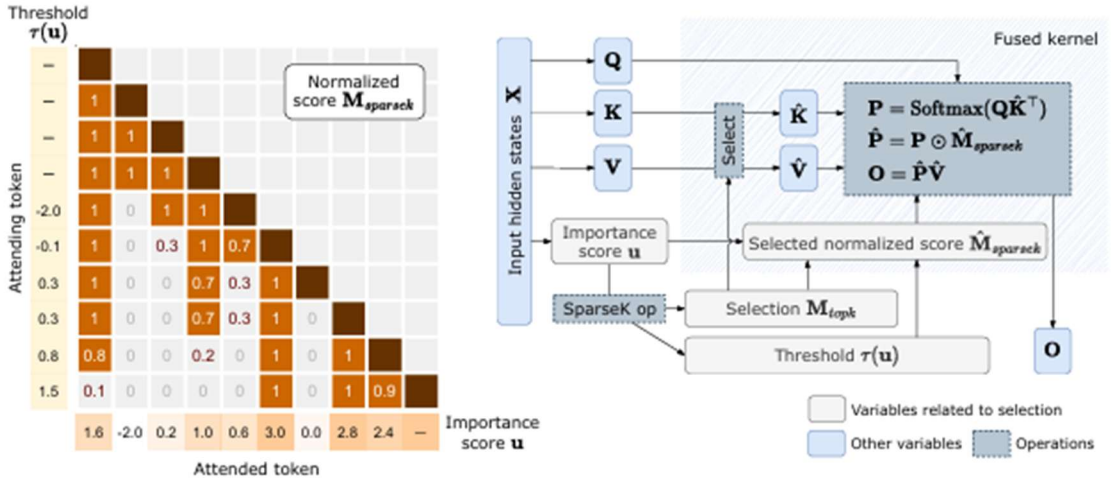


Figure 1: **Left: SPARSEK operation in the attention module.** KV pairs are scored by $\mathbf{u}$. SPARSEK computes a threshold for each query ($\tau(\mathbf{u})$) such that the sum of normalized scores is k, which is 3 in this example. We select top-k KV pairs (orange cells) to perform attention. **Right: the SPARSEK attention module.** We fuse selection and attention in one kernel for efficiency.

Figure 1 (adapted from Lou et al., 2024): SPARSEK attention module. Left: top-k selection via thresholded normalized scores τ(u). Right: fused computation kernel integrates projection-based selection and attention score computation.

Architecture and Innovations:

- A differentiable projection-based sparse operator that approximates top-k key-value selection.
- A fused Triton kernel that merges attention computation and masking for memory-efficient inference.
- Support for **chunk-wise recurrent attention**, making SPARSEK compatible with long-context training and decoding.
- Flexible integration with **sliding window** and **linear attention** mechanisms.

Algorithm 1 Evaluate SPARSEK$(\boldsymbol{z}, k)$

Input: $\boldsymbol{z}$
Sort $\boldsymbol{z}$ as $z_{(1)} \geq \ldots \geq z_{(m)}$
Pre-compute the cumulative sum of $\boldsymbol{z}$
for (u, w) in the descending order **do**
  $\tau \leftarrow$ as in Equation (13)
  if $z_{(w)} > \tau$ and $z_{(u)} \geq \tau + 1$ **then**
    break
  end if
end for
$\boldsymbol{p} \leftarrow \max(\min(\boldsymbol{z} - \tau, \mathbf{1}), \mathbf{0})$
Output: $\boldsymbol{p}$

Algorithm 3 Train with chunk-wise recurrency

Input: $\boldsymbol{X} = [\boldsymbol{X}_1, \boldsymbol{X}_2, \ldots, \boldsymbol{X}_l]$
Initialize a KV cache with scores
for all $\boldsymbol{X}_i$ in $\boldsymbol{X}$ **do**
  $\boldsymbol{O}_i \leftarrow$ attention with $\boldsymbol{X}_i$ and the KV cache
  Add new KV pairs and scores to the cache
  Prune the KV cache to size k by scores
  Stop gradients of the KV cache
end for
Output: $[\boldsymbol{O}_1, \boldsymbol{O}_2, \ldots, \boldsymbol{O}_l]$

Algorithm 2 Evaluate SPARSEK$(\boldsymbol{z}_{1:t}, k)$ at step t from the result of step $t-1$

Input: z_t, min-heaps $\mathcal{F} = \{z_i | z_i \geq \tau(z_{1:t-1}) + 1\}$ and $\mathcal{S} = \{z_i | z_i > \tau(z_{1:t-1})\}$, τ_{t-1} from step $t-1$
~~Sort $\boldsymbol{z}$ as $z_{(1)} \geq \ldots \geq z_{(m)}$~~
~~Pre-compute the cumulative sum of $\boldsymbol{z}$~~
if $z_t \geq \tau(z_{1:t-1}) + 1$ **then**
  Insert z_t into $\mathcal{F}$
end if
if $z_t > \tau(z_{1:t-1})$ **then**
  Insert z_t into $\mathcal{S}$
end if
for (u, w) in the descending order from $(|\mathcal{S}|, |\mathcal{F}|)$ **do**
  $\tau \leftarrow$ as in Equation (13)
  Prune $\mathcal{S}$ and $\mathcal{F}$ with the new τ
  if $z_{(w)} > \tau$ and $z_{(u)} \geq \tau + 1$ **then**
    break
  end if
end for
$\boldsymbol{p} \leftarrow \max(\min(\boldsymbol{z} - \tau, \mathbf{1}), \mathbf{0})$
Output: $\boldsymbol{p}, \mathcal{S}, \mathcal{F}, \tau$

Algorithm 1 – Evaluate SPARSEK(z, k)

Selects top-k values from a score vector z using a soft threshold function. Enables differentiable, trainable sparse selection. Efficiently prunes attention weights using constrained projection.

Algorithm 2 – Incremental SPARSEK($z_{1:t}$, k)

Optimizes the selection process for step-by-step token inputs. Reuses previous computations with two min-heaps, reducing complexity to O(m log m). Essential for streaming and autoregressive tasks.

Algorithm 3 – Chunk-wise Recurrent Training

Allows SPARSEK to operate over long sequences by caching and pruning KV pairs per chunk. Avoids recomputation, enabling memory-efficient training. Compatible with large-scale context extension.

Achieved Results:

Table 1: Perplexity on the OpenWebText held-out set.

Model	Training Context Length		
	1024	4096	8192
Full attention	23.13	21.64	21.86
SW	23.90	23.99	23.10
Linear + SW	23.27	22.97	23.23
Fixed	23.26	22.47	22.86
Random	30.77	34.49	49.76
Hash	26.53	27.42	27.58
GLA	23.29	22.36	24.15
RetNet	24.55	24.50	26.75
SPARSEK + SW	22.85	21.98	21.84
SPARSEK + Linear + SW	22.46	21.55	21.32

Table 1: On OpenWebText, SPARSEK+SW matches or outperforms full attention in perplexity at all sequence lengths, with less memory.

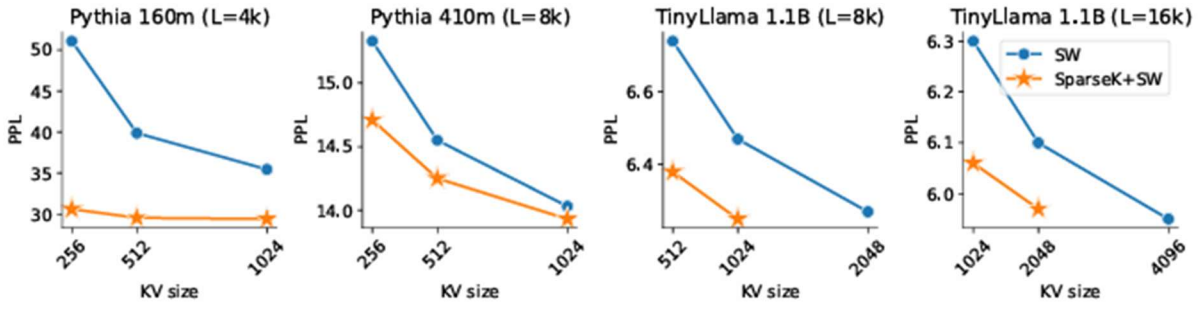
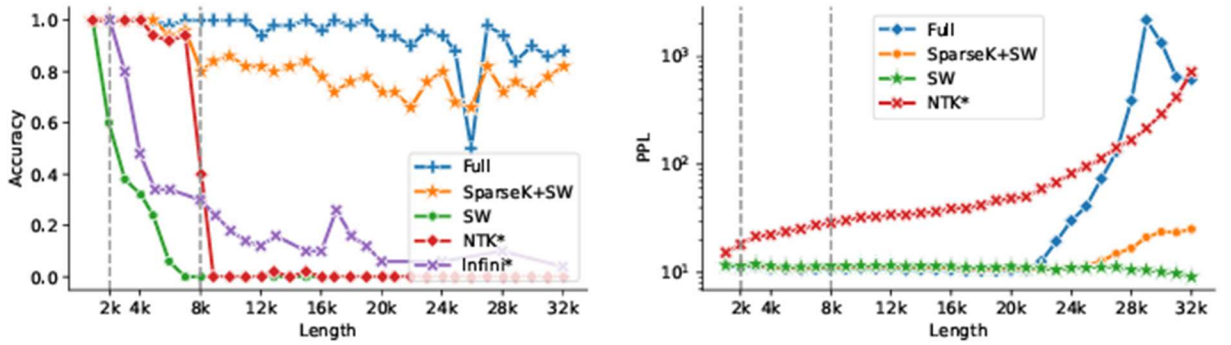


Figure 2: Perplexity on the held-out set of fine-tuned models. L denotes the training context length.

Figure 2: Across four model sizes and training contexts, SPARSEK+SW consistently outperforms SW in perplexity.



(a) Passkey retrieval accuracy.

(b) Perplexity of tokens in different position buckets on the PG19 test set.

Figure 3: Length extrapolation results. * denotes that the method is training-free. 2,048 is the context length of the original model. 8,192 is the context length in fune-tuning.

Figure 3a: At 32K tokens, SPARSEK+SW maintains high retrieval accuracy, outperforming Infini and NTK.

Figure 3b: SPARSEK+SW avoids the perplexity explosion seen in NTK and full attention at long sequence lengths, particularly in the tail segments.

Table 2: Results on LongBench. * denotes that the method is training-free. † We use 512 globel (heavy-hitter) KV cache and 512 local KV cache in H2O.

Model	Single-Doc QA			Multi-Doc QA			Summarziation			Few-shot Learning			Synthetic		Code		Avg.
	NQA	Qspr	MulFi	HQA	WMQA	Musq	GRpt	QMSM	MulN	TREC	TriQA	SMSM	PsgC	PsgR	LCC	Repo	
NTK* $w = 8192$	4.34	10.30	14.54	6.49	9.19	3.49	11.77	7.84	3.62	49.5	55.17	22.66	1.21	3.38	52.19	48.90	19.04
Full $w = 8192$	3.95	13.07	13.16	6.81	10.77	3.51	15.17	6.12	8.30	61.00	65.15	26.02	0.39	2.37	56.72	50.36	21.42
Full H2O†	7.66	9.33	13.73	6.36	10.23	3.26	12.10	7.00	0.87	51.00	54.92	18.31	2.39	2.62	41.66	43.24	17.79
SW $w = 1024$	1.34	8.69	5.41	2.76	4.46	0.48	11.78	4.25	2.39	25.50	13.43	5.33	2.3	0.50	52.22	27.50	10.52
SparseK+SW $k = w = 512$	5.19	14.29	13.24	6.85	9.21	3.83	14.11	5.97	5.85	55.00	52.06	24.79	0.61	2.61	53.90	50.89	19.90

Table 2: On LongBench (QA, summarization, few-shot, code), SPARSEK+SW achieves competitive accuracy while remaining memory-efficient.

Key Contributions:

- Introduces **SPARSEK**, a learnable and hardware-efficient sparse attention operator.
- Fuses sparse selection and attention into a single Triton kernel for GPU acceleration.
- Scales to **32K+ token sequences** with constant-memory inference, outperforming static sparse baselines.
- Enables differentiable, projection-based sparsity that generalizes to long sequences without retraining.

Why it matters:

SPARSEK proves that **attention sparsity can be both adaptive and hardware-friendly**, overcoming the rigidity of fixed pruning methods. Its **end-to-end trainability** and **memory efficiency** make it ideal for real-world deployment in long-context applications like document processing, code generation, and retrieval-based systems.

Where to use:

SPARSEK is best suited for:

- **Long-context autoregressive decoding**
- **Document-level summarization**
- **Code generation**
- **Retrieval-augmented generation (RAG)** Its compatibility with **PyTorch**, **Triton**, and **chunk-wise training** makes it deployable in both academic and industrial pipelines for high-throughput, memory-sensitive inference.

1.3 Inference-Time Filtering and Approximate Attention

Focuses on training-free methods that reduce compute via online approximations.

1.3.1 SpargeAttn: Accurate Sparse Attention for Any Model Inference

SpargeAttn is a runtime-efficient sparse attention mechanism designed to accelerate inference across vision, language, and multimodal generation models. It dynamically predicts and applies sparse attention masks at the block level, requiring no model retraining and preserving output fidelity. Unlike static heuristics or handcrafted pruning,

SpargeAttn leverages runtime attention statistics to drive fine-grained, hardware-aware skipping of unnecessary computations.

Core Methodology

SpargeAttn dynamically predicts a sparse attention pattern based on runtime input statistics and executes the pruned computation using fused GPU kernels. The framework consists of three tightly integrated stages (**Figure 3**):

- **Step 1: Sparse Block Prediction** — Computes cosine similarity between Q and K blocks, aggregates statistics through mean pooling, and selects the top candidates using softmax filtering over cumulative scores.
- **Step 2: Sparse Block Masking** — Incorporates a GPU warp-local gating mechanism that skips Q-K block matmuls if their maximum score falls below a global relevance threshold.
- **Step 3: Warp-Level Softmax** — Applies FlashAttention-style fused softmax computation over the selected blocks for efficient inference.

This procedure produces input-adaptive sparsity patterns tailored to each layer and batch without requiring architectural modifications. The entire pipeline is implemented as a fused kernel, minimizing memory movement and maximizing throughput.

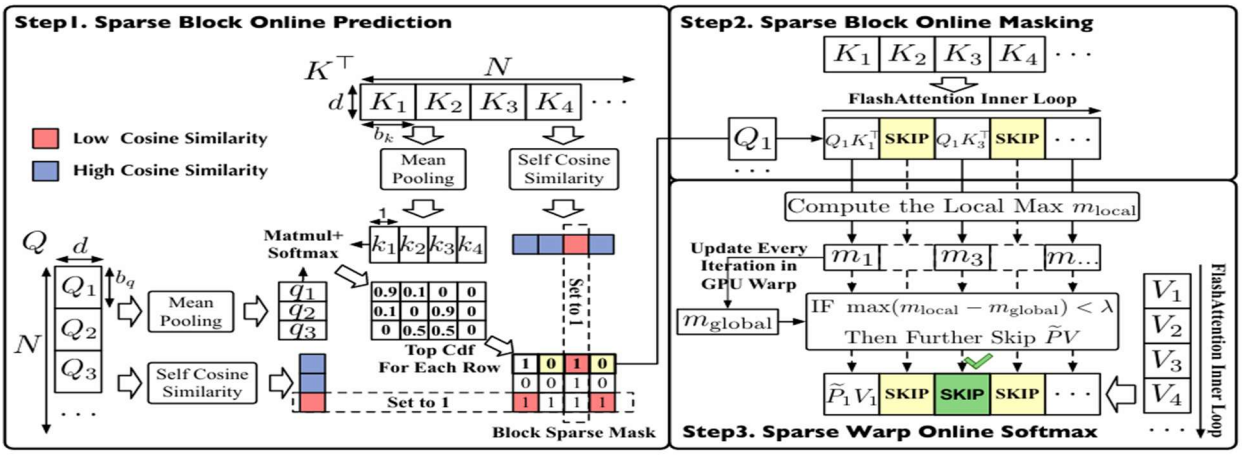


Figure 3. Workflow of SpargeAttn.

Figure 3: End-to-end pipeline of SpargeAttn: prediction → masking → softmax. Enables real-time block skipping on GPU.

Implementation Details

Algorithm 1 fuses quantization, filtering, and sparse matmul into a single GPU-efficient routine:

- Quantizing input tensors per block and computing local cosine similarities.
- Applying thresholded softmax filtering to retain only high-utility attention blocks.
- Performing selective matmuls in GPU warps, followed by sparse softmax aggregation.

This enables fine-grained pruning at inference time with negligible overhead, even for long sequences. Block prediction and softmax operations are parallelized across warps for hardware efficiency.

Algorithm 1 Implementation of `SpargeAttn`.

Input: Matrices $Q(\text{FP16}), K(\text{FP16}), V(\text{FP16}) \in \mathbb{R}^{N\times d}$, block size b_q, b_{kv}, count of GPU Warps c_w, hyper-parameters τ, θ, and λ.
Divide Q to $T_m = N/b_q$ blocks $\{Q_i\}$; divide K, V to $T_n = N/b_{kv}$ blocks $\{K_i\}$ and $\{V_i\}$.
$\hat{Q}_i, \hat{K}_j, \delta_Q, \delta_K = \text{Quant}(Q_i, K_j)$; // per-block quantization in SageAttention.
$q = \{q_i\} = \{\text{mean}(Q_i, \text{axis} = 0)\}$; $k = \{k_j\} = \{\text{mean}(K_j, \text{axis} = 0)\}$;
$\hat{S} = qk^\top$; $s_{qi} = \text{CosSim}(Q_i)$; $s_{kj} = \text{CosSim}(K_j)$; $\hat{S}[:, j] = -\infty$, If $s_{kj} < \theta$;
$\hat{P}[i] = \text{Softmax}(\hat{S}[i])$; $M[i,:] = \text{TopCdf}(\hat{P}[i], \tau)$; $M[i,:] = 1$, If $s_{qi} < \theta$; $M[:, j] = 1$, If $s_{kj} < \theta$;
for $i = 1$ **to** T_m **do**
  Load $\hat{Q}_i$ and $\delta_Q[i]$ into a SM ;
  for j in $[1, T_n]$ **do**
    if $M[i, j]! = 0$ **then**
      Load $\hat{K}_j$, $\hat{V}_j$, and $\delta_K[j]$ into the SM ;
      $S_{ij} = \text{Matmul}(\hat{Q}_i, \hat{K}_j^T) \times \delta_Q \times \delta_K$; // dequantization of SageAttention.
      $m_{\text{local}} = \text{rowmax}(S_{ij})$; $m_{ij} = \max(m_{i,j-1}, m_{\text{local}})$; $\widetilde{P}_{ij} = \exp(S_{ij} - m_{ij})$; $l_{ij} = e^{m_{i,j-1} - m_{ij}} + \text{rowsum}(\widetilde{P}_{ij})$;
      $i_w = \text{range}(c_w)$; $I_w = [\frac{i_w * b_q}{c_w} : \frac{(i_w+1) * b_q}{c_w}]$;
      if $\max(m_{\text{local}}[I_w] - m_{ij}[I_w]) < \lambda$ **then**
        $O_{ij}[I_w] = \text{diag}(e^{m_{i,j-1}[I_w] - m_{ij}[I_w]})^{-1} O_{i,j-1}[I_w] + \text{Matmul}(\widetilde{P}_{ij}[I_w], V_j)$; // Paralleled by c_w warps.
      end if
    end if
  end for
  $O_i = \text{diag}(l_{i,T_n})^{-1} O_{i,T_n}$; Write O_i ;
end for
return $O = \{O_i\}$;

Algorithm 1: Implements joint block pruning and softmax via fused GPU execution with local/global gating.

Hardware and Speed Gains

SpargeAttn achieves significant speedups over full attention while maintaining quality:

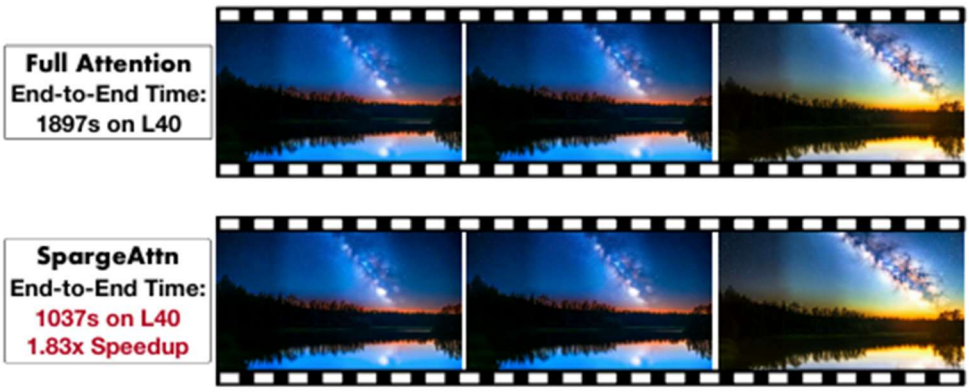


Figure 1. SpargeAttn can achieve 1.83x speedup on Mochi on L40 GPU, with no video quality loss.

Figure 1: Achieves a 1.83× speedup on *Mochi* (22K token generation) with no perceptual loss in video quality.

Table 2. End-to-end generation latency using SpargeAttn.

Model	GPU	Original	SageAttn	SpargeAttn
CogvideoX	RTX4090	87 s	68 s	**53 s**
Mochi	L40	1897 s	1544 s	**1037 s**
Llama3.1 (24K)	RTX4090	4.01 s	3.53 s	**2.6 s**
Llama3.1 (128K)	L40	52 s	42s	**29.98 s**

Table 2: Demonstrates latency reductions of up to 2× across *CogvideoX*, *LLaMA3.1*, and *Mochi*, outperforming SageAttention and FlashAttention2 baselines.

Table 3. Overhead of sparse block prediction in SpargeAttn.

Sequence Len	Prediction (ms)	Full Attention (ms)	Overhead
8k	**0.251**	6.649	3.78%
16k	**0.487**	26.83	1.82%
32k	**0.972**	106.68	0.911%
64k	**2.599**	424.24	0.612%
128k	**8.764**	1696.2	0.516%

Table 3: Prediction overhead remains under 1% for all tested sequence lengths, confirming runtime viability at scale.

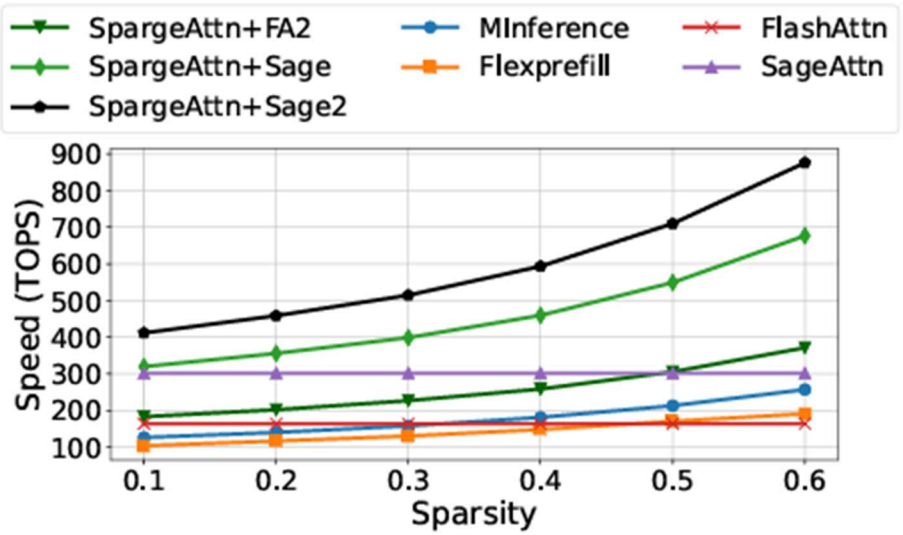


Figure 9. Kernel speed comparison under varying sparsity on RTX4090. Input tensors have a sequence length of 22K and a head dimension of 128. *SpargeAttn+FA2/Sage/Sage2* means deploying our method on FlashAttention2, SageAttention or SageAttention2 (Zhang et al., 2024a).

Figure 9: Attains >700 TOPS on RTX4090 when paired with FlashAttention2, significantly outperforming alternative sparse kernels (e.g., FlexPrefill, Minference).

These results establish SpargeAttn as one of the most hardware-efficient sparse attention operators to date.

Accuracy and Generalization

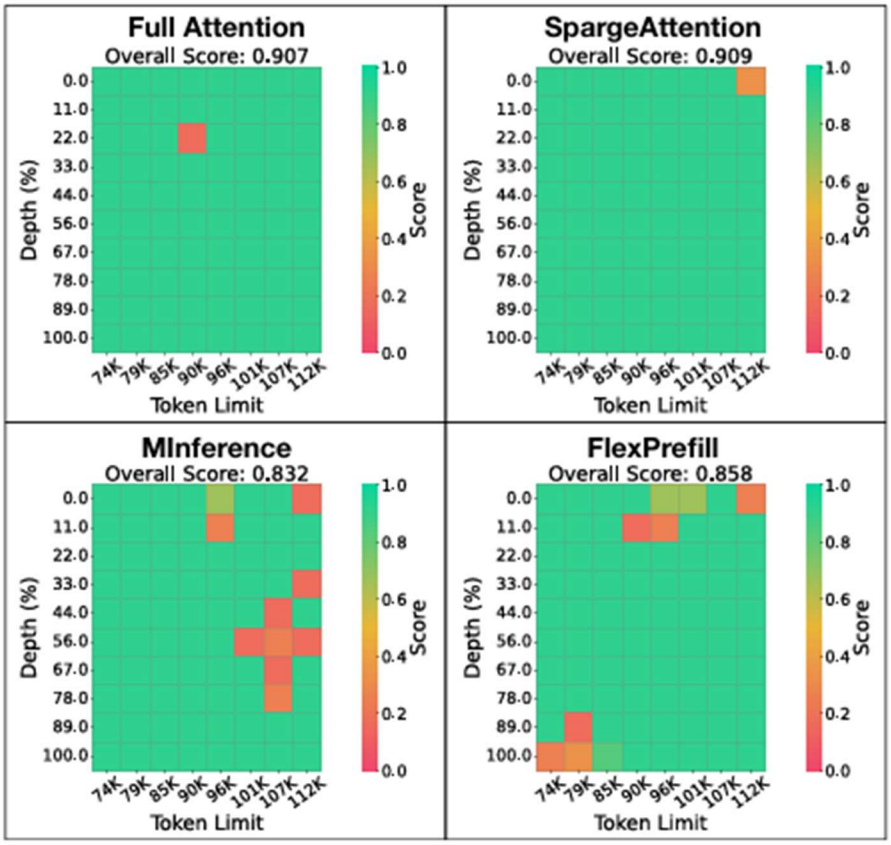


Figure 8. A NeedleInAHaystack comparison example on `Llama3.1`. The sparsity of `SpargeAttn`, MInference, and FlexPrefill is 0.5, 0.5, and 0.54.

Figure 8: On the **NeedleInAHaystack** benchmark, SpargeAttn achieves accuracy on par with full attention (score: 0.909), outperforming FlexPrefill and Minference despite using higher sparsity (0.54).

Table 11. End-to-end metrics on `Llama3.1` in the NeedleInAHaystack task with 16-28K sequence lengths.

Model (seq_len)	**Attention** (Sparsity)	**Speed (TOPS)**↑	**NIAH** ↑
`Llama3.1` (24K)	Full-Attention	156.9	0.838
	Minference (0.5)	122.5	0.635
	FlexPrefill (0.6)	179.6	0.776
	Minference (0.3)	102.3	0.652
	FlexPrefill (0.3)	117.6	0.797
	`SpargeAttn` (0.36)	**443.6**	**0.863**

Table 11: At 24K sequence length, SpargeAttn achieves the **highest accuracy (0.863)** and **2.8× speedup** compared to dense attention.

Table 7. Sparsity increases with sequence length under a constant accuracy bound on `Llama3.1`.

Sequence Len	8K	16K	24K	48K	128K
Sparsity	6.8%	26.4%	35.7%	49.8%	54%

Table 7: Sparsity increases naturally with input length under fixed accuracy targets, reaching **54%** at 128K tokens.

These results highlight SpargeAttn's ability to generalize across modalities and maintain accuracy under tight memory budgets.

Design Insights

SpargeAttn includes several architectural refinements to improve sparsity and fidelity:

Table 4. Effect of permutation on sparsity and accuracy. Sim-q and Sim-k are the average block self-similarity of the query and key.

Method	Sim-q ↑	Sim-k ↑	L1 ↓	Sparsity ↑
Random	0.321	0.019	0.0414	0.048
Rowmajor	0.551	0.390	**0.0307**	0.363
Timemajor	0.514	0.367	0.0342	0.338
HilbertCurve	**0.572**	**0.479**	0.0389	**0.392**

Table 4: Hilbert curve permutations maximize block-level self-similarity, yielding the highest sparsity (0.392) and lowest reconstruction loss.

Table 5. Abalation of self-similarity judge.

Method	VQA-a ↑	VQA-t ↑	FScore ↑
W/o. self-sim Judge	34.664	44.722	1.138
With self-sim Judge	54.179	67.219	1.807

Table 5: A self-similarity judge improves VQA accuracy by 20+ points, confirming the benefit of structured pruning criteria.

Table 6. Analysis of sparsity from M_g and M_{pv}.

Strategy	only M_g	only M_{pv}	$M_g + M_{pv}$
Sparsity	51.2%	27.7%	54%

Table 6: Combining global and predictive sparsity masks achieves the highest compression rate (54%), outperforming either strategy alone.

Summary of Contributions

- **Proposes a fused, input-adaptive sparse attention kernel requiring no model modification or retraining.**
- **Achieves state-of-the-art inference throughput (708 TOPS) and latency reduction with minimal quality degradation.**
- **Demonstrates strong generalization to long contexts and multiple modalities (text, vision, diffusion).**
- **Integrates seamlessly with existing high-performance attention backends (e.g., FlashAttention2).**

Why It Matters

SpargeAttn achieves the **ideal trifecta** for sparse attention in real-world inference:

- **Fast** (over 700 TOPS in hardware tests)
- **Accurate** (matches full attention benchmarks)
- **Practical** (no retraining, deployable on existing models)

It generalizes across domains, supports massive contexts, and requires minimal engineering intervention to adopt.

Where to Use

SpargeAttn is ideal for:

- **Latency-sensitive inference** (e.g., retrieval, summarization)

- **Long-context decoding** (Llama3.1 @ 128K)
- **Vision-language generation** (Flux, StableDiffusion3.5, Mochi)
- **Inference accelerators using FlashAttention2-compatible stacks**

1.4 Memory-Efficient Sparse Kernels (MLP + Projection Layers)

These are not attention-specific but complement attention sparsity in PyTorch-based inference.

SparseGPT introduces a novel, scalable one-shot pruning framework designed to sparsify large generative pretrained transformer (GPT) models without requiring retraining. Addressing the significant memory and compute bottlenecks in deploying massive models such as OPT-175B and BLOOM-176B, SparseGPT demonstrates that 50–60% sparsity can be achieved with minimal accuracy degradation. This enables more feasible inference on single-GPU systems and large-scale deployment of sparsified models across heterogeneous hardware platforms.

Method Overview

SparseGPT frames pruning as a sparse regression problem and solves it via an efficient weight reconstruction strategy. Rather than retraining, it sequentially prunes weights in a layer while minimizing output error, using a fast, batched approximation of Hessian updates. This approach enables sparsity induction even on models with hundreds of billions of parameters. SparseGPT supports unstructured sparsity (e.g., 50%) as well as semi-structured formats such as 2:4 and 4:8 sparsity, while optionally combining pruning with low-bit quantization in a single compression pass.

Implementation and Algorithmic Details

SparseGPT operates through the following steps:

- **Weight and Mask Initialization**: A pruning mask is adaptively generated, targeting a desired sparsity pattern across each layer.
- **OBS-Based Pruning**: SparseGPT employs Optimal Brain Surgeon (OBS)-style pruning, updating the weight matrix to compensate for the pruned entries using a locally computed Hessian inverse.
- **Batchwise and Lazy Updates**: Rather than updating after every weight removal, SparseGPT performs lazy batched updates using efficient block quantization of the error matrix, significantly improving scalability.

- **Adaptive Masking**: Columns are sparsified blockwise based on estimated pruning error, adjusting dynamically to local layer sensitivities.
- **Joint Quantization Option**: Pruned and frozen weights are quantized on-the-fly, enabling further size reduction without retraining.

The algorithmic structure is formally detailed in **Algorithm 1 (SparseGPT Algorithm)**, providing the pseudocode for layer-wise sparsification.

Algorithm 1 The `SparseGPT` algorithm. We prune the layer matrix $\mathbf{W}$ to $p\%$ unstructured sparsity given inverse Hessian $\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$, lazy batch-update blocksize B and adaptive mask selection blocksize B_s; each B_s consecutive columns will be $p\%$ sparse.

$\mathbf{M} \leftarrow \mathbf{1}_{d_{\text{row}} \times d_{\text{col}}}$ *// binary pruning mask*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ *// block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^\top$ *// Hessian inverse information*
for $i = 0, B, 2B, \ldots$ **do**
  for $j = i, \ldots, i + B - 1$ **do**
    if $j \bmod B_s = 0$ **then**
      $\mathbf{M}_{:,j:(j+B_s)} \leftarrow$ mask of $(1-p)\%$ weights $w_c \in \mathbf{W}_{:,j:(j+B_s)}$ with largest $w_c^2 / [\mathbf{H}^{-1}]_{cc}^2$
    end if
    $\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j} / [\mathbf{H}^{-1}]_{jj}$ *// pruning error*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{1} - \mathbf{M}_{:,j}) \cdot \mathbf{E}_{:,j-i}$ *// freeze weights*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ *// update*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ *// update*
end for
$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}$ *// set pruned weights to 0*

Algorithm 1 - Structure of the algorithm

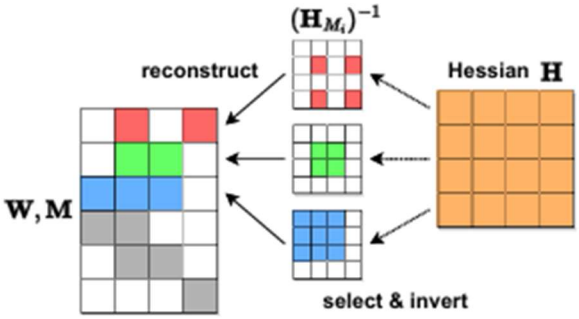


Figure 3. Illustration of the row-Hessian challenge: rows are sparsified independently, pruned weights are in white.

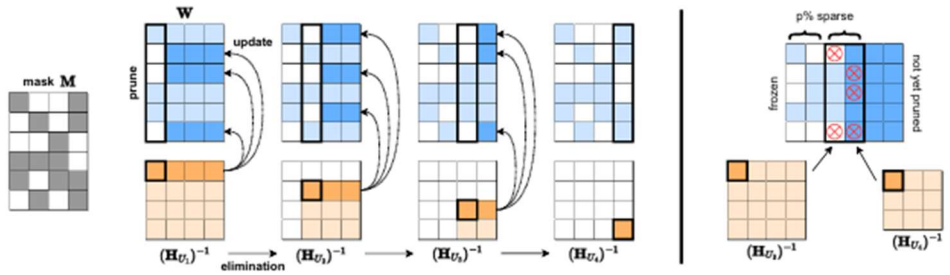


Figure 4. [Left] Visualization of the SparseGPT reconstruction algorithm. Given a fixed pruning mask $\mathbf{M}$, we incrementally prune weights in each column of the weight matrix $\mathbf{W}$, using a sequence of Hessian inverses $(\mathbf{H}_{U_j})^{-1}$, and updating the remainder of the weights in those rows, located to the "right" of the column being processed. Specifically, the weights to the "right" of a pruned weight (dark blue) will be updated to compensate for the pruning error, whereas the unpruned weights do not generate updates (light blue). [Right] Illustration of the adaptive mask selection via iterative blocking.

Figure 3 and **Figure 4** illustrate the challenges in sparse Hessian inversion and depict the core update mechanism, respectively, offering intuitive understanding of the method's computational flow.

Efficiency and Hardware Scaling

SparseGPT exhibits strong scalability across extremely large models:

- On the OPT-175B model, SparseGPT achieves **50% sparsity with minimal perplexity loss** and completes pruning within approximately four hours on a single NVIDIA A100 GPU.
- In scaling tests, SparseGPT consistently induces 50–60% sparsity while maintaining competitive perplexity across model scales from 125M to 175B parameters (**Figure 1**, **Figure 2**, **Figure 5**).

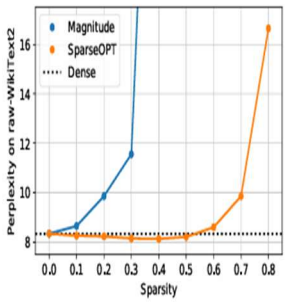


Figure 1. Sparsity-vs-perplexity comparison of SparseGPT against magnitude pruning on OPT-175B, when pruning to different *uniform* per-layer sparsities.

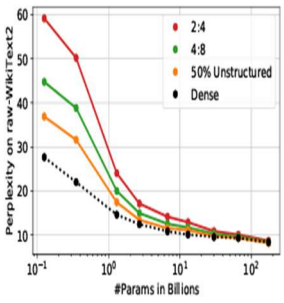


Figure 2. Perplexity vs. model and sparsity type when compressing the entire OPT model family (135M, 350M, …, 66B, 175B) to different sparsity patterns using SparseGPT.

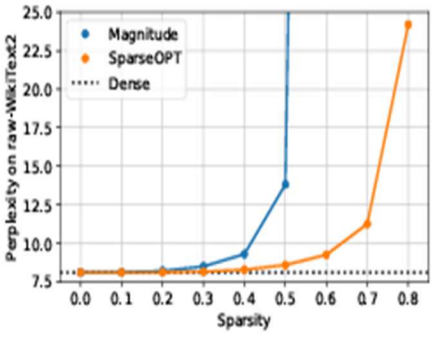


Figure 5. Uniform pruning BLOOM-176B.

Figure 1, 2, 5.

- Notably, larger models (e.g., OPT-66B, OPT-175B) are easier to sparsify, achieving higher compression ratios compared to smaller ones.

- When combined with 4-bit quantization, SparseGPT outperforms traditional 3-bit quantization in size-to-accuracy efficiency (**Figure 6**).

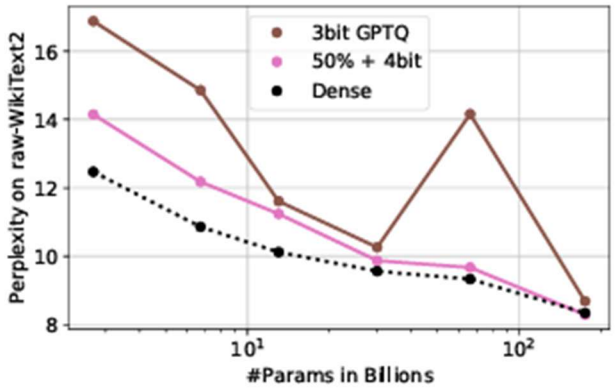


Figure 6. Comparing joint 50% sparsity + 4-bit quantization with size-equivalent 3-bit on the OPT family for $\geq$ 2.7B params.

Figure 6.

Accuracy and Generalization

SparseGPT preserves model accuracy across a range of downstream tasks and evaluation benchmarks:

- On raw WikiText2, SparseGPT achieves perplexities close to the dense model even at 50% sparsity (**Table 1**).

Table 1. OPT perplexity results on raw-WikiText2.

OPT - 50%	125M	350M	1.3B
Dense	27.66	22.00	14.62
Magnitude	193.	97.80	1.7e4
AdaPrune	58.66	48.46	32.52
SparseGPT	**36.85**	**31.58**	**17.46**

OPT	Sparsity	2.7B	6.7B	13B	30B	66B	175B
Dense	0%	12.47	10.86	10.13	9.56	9.34	8.35
Magnitude	50%	265.	969.	1.2e4	168.	4.2e3	4.3e4
SparseGPT	50%	**13.48**	**11.55**	**11.17**	**9.79**	**9.32**	**8.21**
SparseGPT	4:8	14.98	12.56	11.77	10.30	9.65	8.45
SparseGPT	2:4	17.18	14.20	12.96	10.90	10.09	8.74

Table 1.

- In zero-shot task evaluations, SparseGPT maintains performance levels comparable to the dense baseline across Lambada, PIQA, ARC-e, ARC-c, and StoryCloze datasets (**Table 2**).

Table 2. ZeroShot results on several datasets for sparsified variants of OPT-175B.

Method	Spars.	Lamb.	PIQA	ARC-e	ARC-c	Story.	**Avg.**
Dense	0%	75.59	81.07	71.04	43.94	79.82	**70.29**
Magnitude	50%	00.02	54.73	28.03	25.60	47.10	**31.10**
SparseGPT	50%	78.47	80.63	70.45	43.94	79.12	**70.52**
SparseGPT	4:8	80.30	79.54	68.85	41.30	78.10	**69.62**
SparseGPT	2:4	80.92	79.54	68.77	39.25	77.08	**69.11**

Table 2.

- The method proves resilient against calibration data variance and hyperparameter shifts, demonstrating robustness in one-shot pruning scenarios.

Design Insights

SparseGPT incorporates several innovations to enable its scalability:

- **OBS-inspired Sparse Updates**: Sequential local updates using Hessian inverses provide superior error compensation without needing global retraining.
- **Batching and Lazy Propagation**: Efficient memory management strategies allow handling of extremely large models by amortizing computational costs.
- **Row-Wise Hessian Management**: Sharing computations across rows enables approximate yet effective weight reconstructions with tractable overhead.
- **Flexible Masking**: Adaptive blockwise pruning permits fine-grained control over sparsity patterns, making SparseGPT compatible with hardware-aware formats like 2:4 or 4:8 sparsity.

Summary of Contributions

- Developed SparseGPT, a scalable one-shot pruning framework for large GPT models.
- Achieved 50–60% sparsity without fine-tuning, across models ranging from 135M to 175B parameters.
- Introduced fast approximate reconstruction leveraging localized Hessian inversion.
- Enabled integration of pruning and quantization in a unified compression pass.
- Delivered memory-efficient, high-throughput inference feasibility for 100B+ parameter models.

Why it Matters

SparseGPT eliminates the need for costly retraining when inducing sparsity in massive language models. It enables inference and deployment of sparsified models on single-GPU systems and resource-constrained platforms. By maintaining near-dense model quality while reducing memory footprint and computational load, SparseGPT advances practical scaling strategies for real-world LLM deployment.

Where to Use

SparseGPT is ideally suited for:

- Production-scale LLM inference pipelines requiring memory and compute efficiency.
- Scenarios combining pruning and quantization for aggressive compression.
- Post-training optimization of proprietary or open-source LLMs for deployment on limited hardware.
- Applications demanding low-latency or high-throughput serving of sparsified LLMs.

Section 2: Computational Complexity Improvements, Parallelization Strategies, and Sparse Matrix Optimizations

MIXED SPARSITY TRAINING: ACHIEVING 4× FLOP REDUCTION FOR TRANSFORMER PRETRAINING

Large-scale Transformer pretraining demands substantial computational resources, often constrained by hardware limitations. To address these challenges, Mixed Sparsity Training (MST) introduces a dynamic sparsification approach that targets both fully-connected and attention layers during training, enabling up to a 4× reduction in floating point operations (FLOPs) without compromising model quality. This method aims to bridge the gap between theoretical sparsity and practical training acceleration for trillion-parameter scale models.

Method Overview

MST dynamically adjusts the sparsity patterns of both the fully connected and self-attention layers throughout training. It divides the training process into three distinct phases—**Warm-up**, **Ultra-Sparsification**, and **Restoration**—where the sparsity level and mask topology evolve to maximize FLOP reduction while preserving model expressivity.

Importantly, MST integrates a hybrid sparse attention mechanism that modulates attention mask sparsity over time.

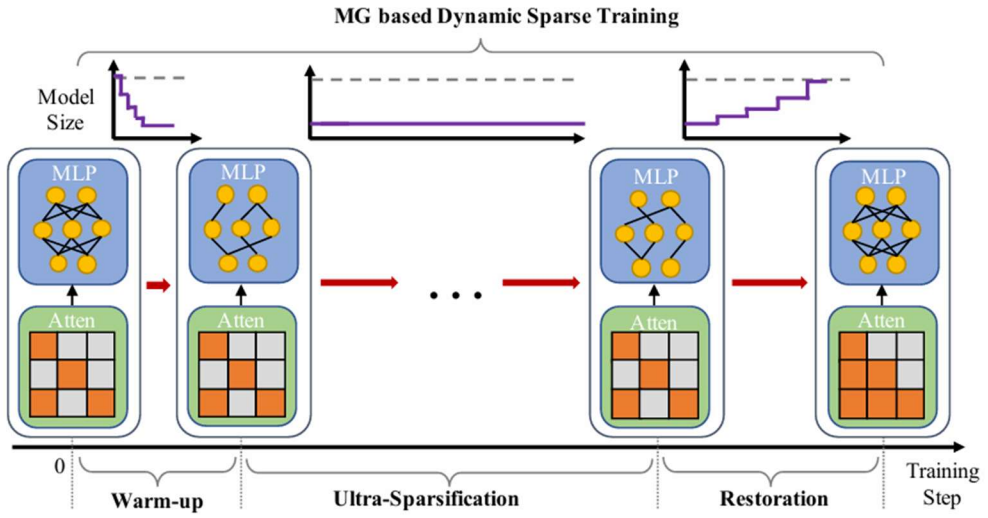


Figure 2: The sparsity variation of MST includes three phases: warm-up, ultra-sparsification and restoration. SV is combined with MG-based dynamic sparse training and HSA during the training.

Figure 2 – Illustration of MST's phased training structure showing transitions between warm-up, ultra-sparsification, and restoration

---

Implementation and Algorithmic Details

The MST framework operates through:

- **Topology Evolution:** Gradually updating the sparsity masks of network weights through a method called Mask Growth (MG), which selectively regrows important parameters based on dynamic importance criteria.
- **Sparse Attention Masking:** Using hybrid attention masks with adjustable stride lengths, MST allows flexible sparsification of attention heads across the training timeline.
- **Piecewise Cosine Annealing:** Regulating the update fraction of growing connections to smooth out sparsity transitions during ultra-sparsification and restoration phases.

Algorithm 1 Topology Evolution by MG.

θ_l, N_l, s_l, s'_l: parameters, number of parameters, current and target sparsity of layer l.
for each layer l **do**
  $N_{\text{prune}} = \zeta(1 - s_l)N_l$
  $N_{\text{grow}} = N_{\text{prune}} + (s'_l - s_l)N_l$
  $N_{\text{rand}} = \lfloor N_{\text{grow}} R \rfloor$
  $N_{\text{grad}} = N_{\text{grow}} - N_{\text{rand}}$
  $\mathbb{I}_{\text{prune}} = \text{ArgTopK}(-|\theta_l \odot M_{\theta_l}|, N_{\text{prune}})$
  $\mathbb{I}_{\text{grad\_grow}} = \text{ArgTopK}_{i \notin \theta_l \odot M_{\theta_l} \setminus \mathbb{I}_{\text{prune}}}(|\nabla_{\theta_l} L|, N_{\text{grad}})$
  $\mathbb{I}_{\text{rand\_grow}} = \text{RandK}_{i \notin \theta_l \odot M_{\theta_l} \setminus (\mathbb{I}_{\text{drop}} \cup \mathbb{I}_{\text{grad\_grow}})}(N_{\text{rand}})$
  $\mathbb{I}_{\text{grow}} = \mathbb{I}_{\text{grad\_grow}} \cup \mathbb{I}_{\text{rand\_grow}}$
  Update M_{θ_l} according to $\mathbb{I}_{\text{prune}}$ and $\mathbb{I}_{\text{grow}}$
  $\theta_l \leftarrow \theta_l \odot M_{\theta_l}$
end for

Algorithm 1 – Pseudocode of the Mask Growth (MG) topology update algorithm governing sparsity evolution

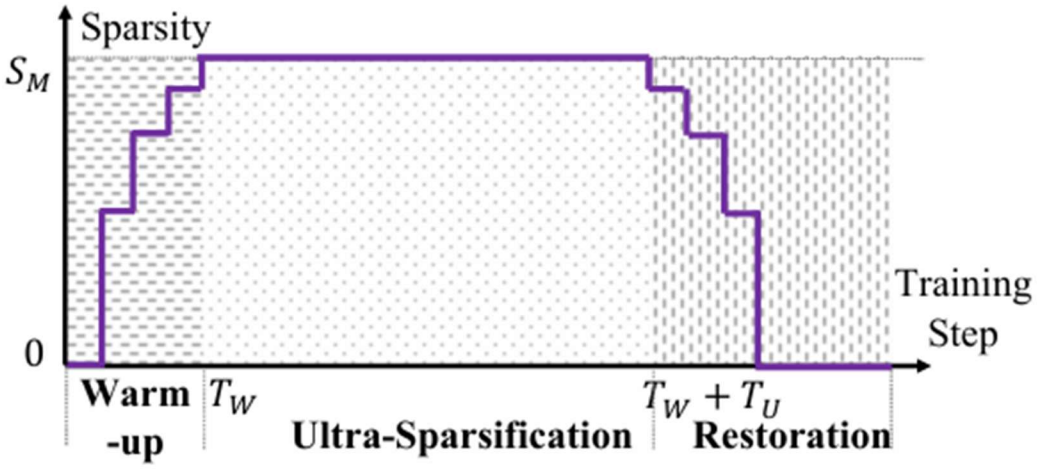


Figure 3: Sparsity variation.

Figure 3 – Sparsity variation curve over training steps, showing how MST dynamically modulates model sparsity

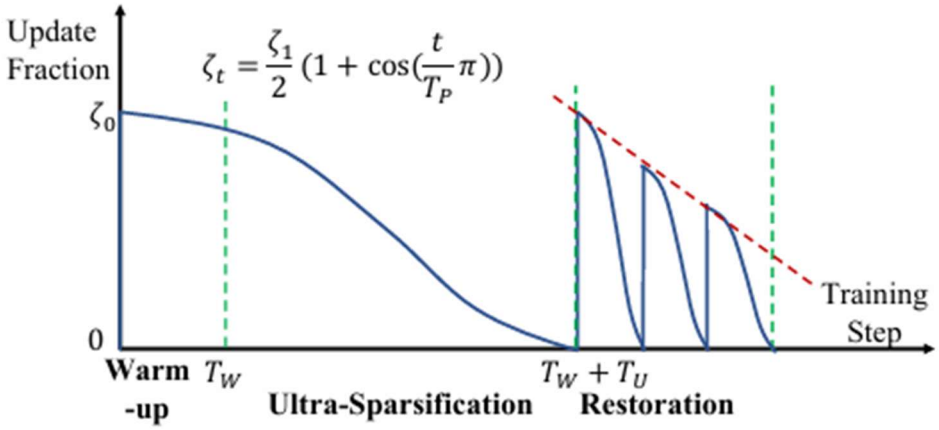


Figure 4: Piecewise cosine annealing.

Figure 4 – Piecewise cosine annealing curve describing the update scheduling strategy for regrowth

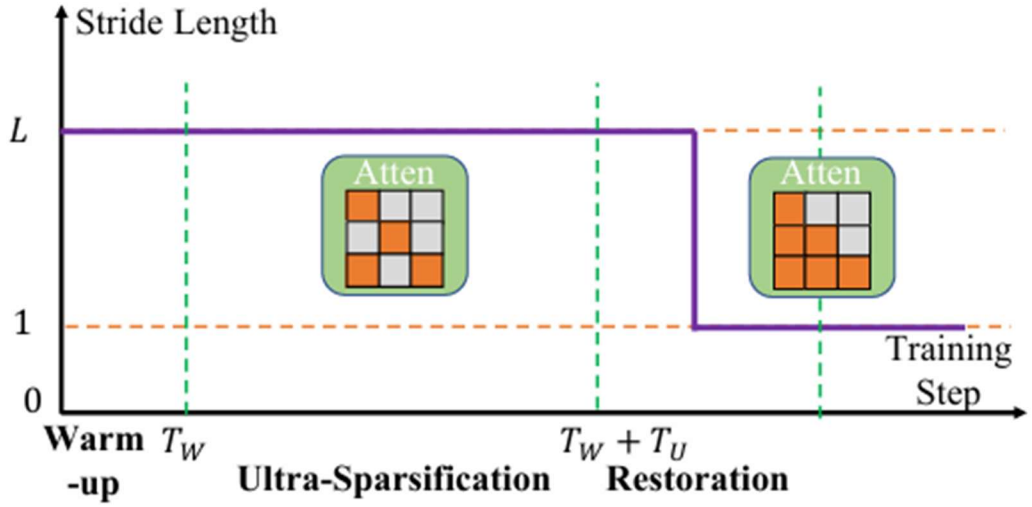


Figure 6: Stride length of attention mask in hybrid sparse attention. Notice that when the stride length equals 1, the model utilizes a dense mask.

Figure 6 – Diagram illustrating the stride-based hybrid sparse attention mechanism applied across training phases

Efficiency and Hardware Scaling

Empirical results demonstrate that MST achieves substantial FLOP reductions during pretraining:

- **4× FLOP reduction** compared to dense training on GPT-2 scale models.
- MST outperforms prior dynamic and static sparsity techniques like Pixelated Butterfly, SPDF, and RigL.
- Demonstrates effective scalability on models from 100M to 1.5B parameters.

Table 1: Comparison of different training techniques for saving transformer pretraining FLOPS.

Name	Methods	Models	Reduction
DynSparse [30]	Dynamic Sparsification	BERT	2×
LiGO [35]	Layer Growth	BERT	2.2×
Monarch [32]	Block Sparsification	GPT-2, BERT	2.2×
SPDF [36]	Static Sparsification	GPT-2, GPT-3	2.5×
Pixelated Butterfly [31]	Butterfly Sparsification	GPT-2	2.6×
MST (Ours)	Dynamic Sparsification	GPT-2	**4**×

Table 1 – Comparative summary of FLOP reduction rates across dynamic and structured sparsification methods, highlighting MST's superior efficiency

Table 3: Performance comparison of different pretraining methods on GPT-2.

Alg.	FLOPs (G)	Zero-shot						Few-shot	
		LAMBADA (ACC)	LAMBADA (PPL)	WikiText2 (PPL)	PTB (PPL)	WikiText103 (PPL)	1BW (PPL)	RTE (ACC)	MRPC (ACC)
Dense	847.8	60.74	13.22	34.89	34.06	36.29	44.16	52.98	71.26
Tiny	212.7	53.29	31.91	58.00	58.71	60.78	71.96	55.87	71.63
SS	267.7	50.59	54.77	83.98	75.98	88.00	99.44	50.00	68.50
RigL	267.7	55.75	25.81	43.00	48.14	44.82	62.31	48.83	67.22
MST	219.4	60.54	13.67	33.33	34.64	34.95	45.90	52.62	70.96

Table 3 – Performance-FLOP tradeoff showing that MST maintains high task accuracy while significantly lowering computation cost

---

Accuracy and Generalization

Despite aggressive sparsification, MST preserves model accuracy:

- Maintains competitive validation perplexity on datasets such as WikiText2 and PTB.
- Shows strong robustness across pretraining phases, avoiding catastrophic forgetting during sparsification.
- Consistently outperforms structured sparsity baselines on zero-shot and few-shot generalization tasks.

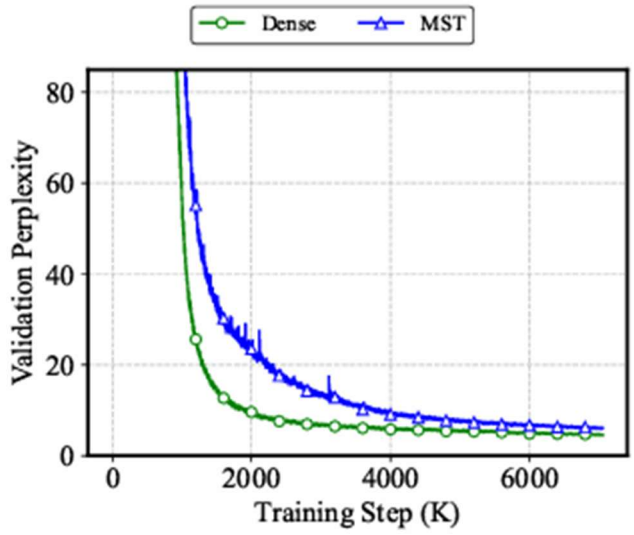


Figure 7: Perplexity of different pre-training methods on BERT.

Figure 7 – Validation perplexity curves comparing dense training and MST on BERT

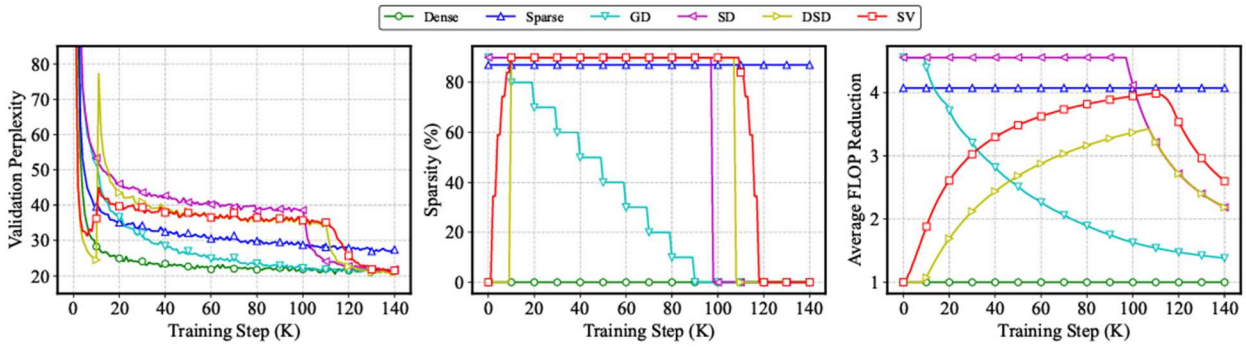


Figure 8: Ablation study on different sparsity variation patterns.

Figure 8 – Ablation study of sparsity schedule variations demonstrating MST's advantage over alternative sparsification dynamics

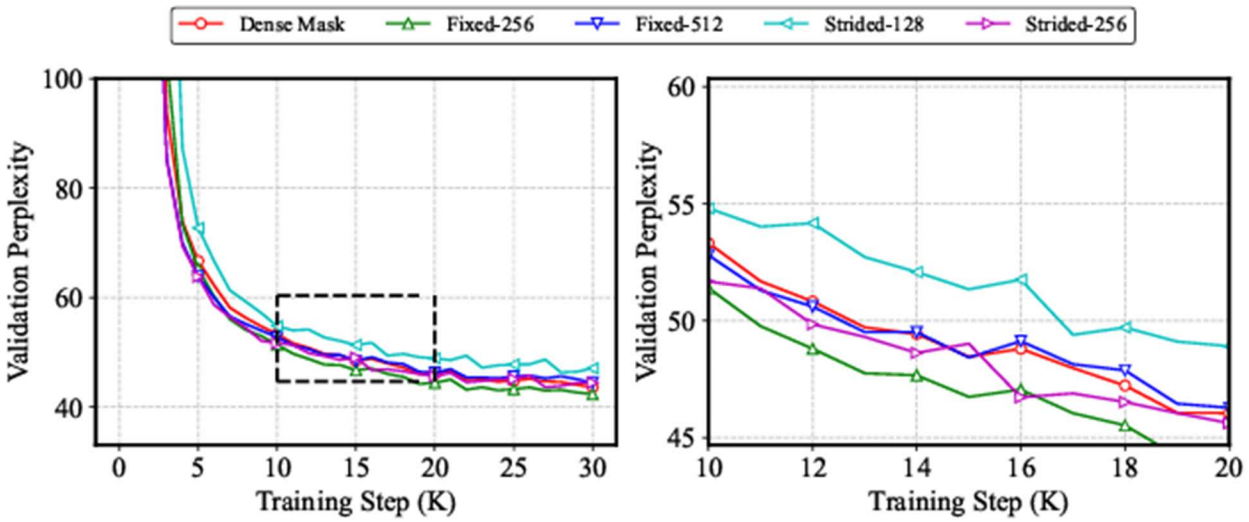


Figure 11: Validation perplexity of different attention patterns.

Figure 11 – Comparison of validation perplexity under different fixed and strided attention mask patterns

---

Design Insights

Key architectural strategies that underpin MST's effectiveness include:

- **Dynamic Hybrid Sparse Attention:** Strided attention masks adaptively transition between dense and sparse modes to balance expressiveness and computational savings.
- **Three-Phase Sparsity Management:** Structuring training into warm-up, ultra-sparsification, and restoration phases prevents information bottlenecks during aggressive pruning.
- **Topology-Guided Growth:** Rather than fixing sparsity patterns at initialization, MST continuously evolves them to align with model learning dynamics.

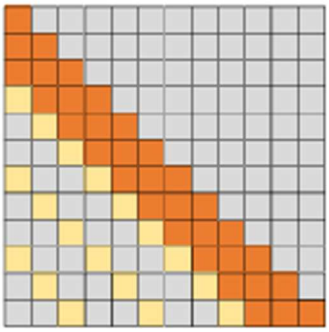


Figure 5: Strided attention with stride length $l = 3$.

Figure 5 – Example of a strided attention mask with stride 3, used during MST training

---

Summary of Contributions

- Introduces a **dynamic sparsification framework** that targets both fully connected and attention layers.
- Proposes a **phased training paradigm** that modulates sparsity over time to maximize FLOP savings and stability.
- Develops a **hybrid sparse attention strategy** with adjustable stride lengths for long-range dependency preservation.
- Demonstrates **4× FLOP reduction** with minimal impact on model performance across multiple benchmarks.

OATS: OUTLIER-AWARE PRUNING THROUGH SPARSE AND LOW RANK DECOMPOSITION

Transformer models continue to grow in size, posing challenges in memory and computation for deployment in real-world settings. OATS (Outlier-Aware Sparsification) proposes a method for compressing large language models by selectively pruning weights and reconstructing their impact using a low-rank decomposition. By jointly applying unstructured pruning and low-rank correction, OATS significantly reduces model size and

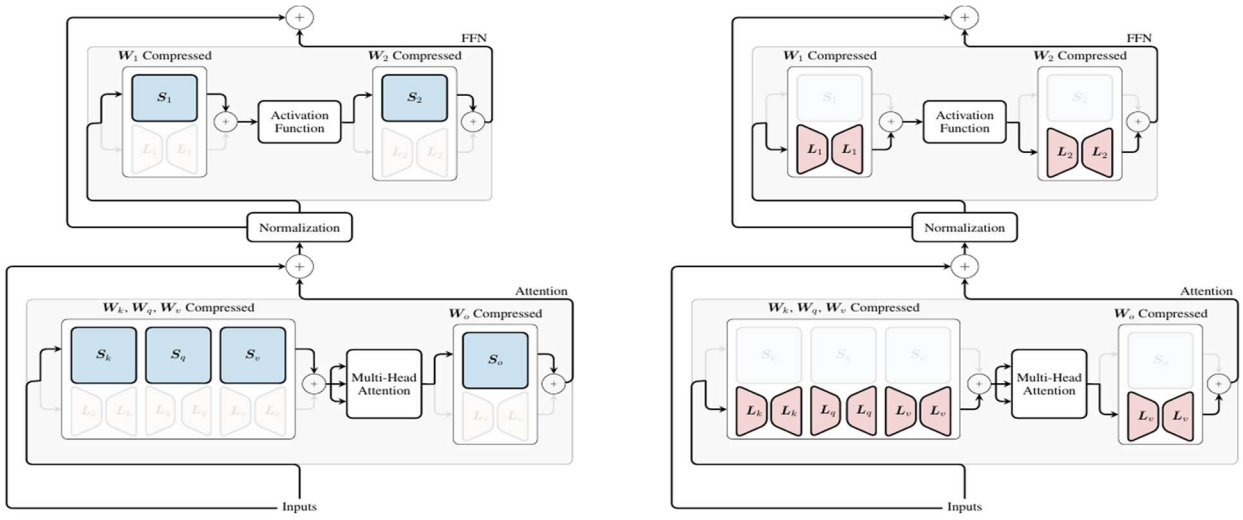
compute requirements while preserving accuracy across zero-shot and few-shot benchmarks. This approach targets practical, scalable compression without extensive retraining.

Method Overview

OATS identifies the critical rows in the weight matrices that contribute most significantly to activations (outliers) and applies a hybrid compression scheme:

- **Sparse Pruning:** Removes a significant fraction of the weights.
- **Low-Rank Correction:** Reconstructs the impact of removed weights using low-rank approximations.

The method incrementally prunes weights by alternating between sparse updates and low-rank refinements, governed by an alternating thresholding algorithm. This dual-structure captures the benefits of both sparsity and low-rank modeling to maintain model performance even at aggressive compression levels.



(a) Sparse terms only

(b) Low-rank terms only

Figure 3: A visualization of how the attention rollout is computed to isolate the contribution of the sparse terms versus low-rank terms given by the OATS algorithm.

Figure 3 *here: Diagram showing how sparse and low-rank decompositions are applied to layers.*

Implementation and Algorithmic Details

The OATS algorithm operates in two primary phases:

1. **Alternating Thresholding (Algorithm 1):**
    - Iteratively prunes weight matrices based on importance.
    - Computes a low-rank approximation using truncated SVD (Singular Value Decomposition).
2. **Outlier-Aware Pruning (Algorithm 2):**
    - Decomposes layer weights into sparse and low-rank parts.
    - Dynamically adjusts pruning rates based on layer activations and model structure.

The sparse and low-rank components are combined during inference to approximate the original computation with minimal quality degradation.

Algorithm 1 ALTERNATINGTHRESHOLDING

Inputs:
Weight Matrix: $\boldsymbol{W} \in \mathbb{R}^{d_{out} \times d_{in}}$
Iterations: N
Rank: r
Nonzeros: k

Procedure:
$\boldsymbol{S} = \boldsymbol{0}$
for $t = 1$ to N **do**
$\boldsymbol{L} = \text{TRUNCATEDSVD}(\boldsymbol{W} - \boldsymbol{S}, r)$
$\boldsymbol{S} = \text{HARDTHRESHOLD}(\boldsymbol{W} - \boldsymbol{L}, k)$
end for
return: $\boldsymbol{S}, \boldsymbol{L}$

Algorithm 2 OATS

Inputs:
Layer Inputs Propagated Through Prior Compressed Layers: $\boldsymbol{X}^{\ell} \in \mathbb{R}^{B \times d_{in}}$
Layer Matrix: $\boldsymbol{W}^{\ell} \in \mathbb{R}^{d_{out} \times d_{in}}$
Compression Rate: ρ
Rank Ratio: κ
Iterations: N

Procedure:
$r \leftarrow \left\lfloor \kappa \cdot (1-\rho) \cdot \frac{d_{out} \cdot d_{in}}{d_{out} + d_{in}} \right\rfloor$, $k \leftarrow \lfloor (1-\kappa) \cdot (1-\rho) \cdot d_{out} \cdot d_{in} \rfloor$
$\boldsymbol{D} \leftarrow \sqrt{\text{diag}(\boldsymbol{X}^{\top}\boldsymbol{X})}$
$\boldsymbol{L}, \boldsymbol{S} \leftarrow \text{ALTERNATINGTHRESHOLDING}(\boldsymbol{W}\boldsymbol{D}, N, r, k)$
$\boldsymbol{W} \leftarrow (\boldsymbol{L} + \boldsymbol{S})\boldsymbol{D}^{-1}$
return: $\boldsymbol{X}^{\ell+1} \leftarrow \boldsymbol{X}^{\ell}\boldsymbol{W}^{\top}$

***Algorithms 1, 2**: AlternatingThresholding and OATS procedures.*

Efficiency and Hardware Scaling

OATS demonstrates substantial speedups across multiple hardware setups:

- Achieves **up to 2.06× throughput improvement** compared to dense baselines at 50% compression.
- Maintains high-speed token generation rates even under aggressive sparsification.

Compression	Method	Throughput	Speedup
0%	Dense	4.03	1.00×
30%	Unstructured Pruning	4.32	1.07×
	OATS	**5.58**	**1.38**×
40%	Unstructured Pruning	5.08	1.26×
	OATS	**6.86**	**1.73**×
50%	Unstructured Pruning	7.16	1.78×
	OATS	**8.31**	**2.06**×

Table 7: Comparison of throughput (tokens/second) and speedup achieved through OATS and unstructured pruning methods relative to their dense counterparts.

***Table 7** here: Speedup and throughput results comparing OATS against unstructured pruning methods.*

Accuracy and Generalization

Extensive evaluation across Phi-3 and Llama-3 model families shows:

- Minimal degradation in five-shot, zero-shot, and perplexity metrics even at **50% compression**.
- Outperforms previous methods like SparseGPT, Wanda, and DSNoT at comparable sparsity.

Compression	Method	Phi-3		Llama-3	
		Mini (3.8B)	Medium (14B)	8B	70B
0%	Dense	70.34	76.78	64.97	79.63
30%	SparseGPT	68.31	74.12	64.25	78.28
	Wanda	67.63	75.18	63.67	**79.15**
	DSNoT	68.02	75.13	63.72	79.00
	OATS	**68.84**	**76.15**	**65.22**	78.47
40%	SparseGPT	63.47	72.42	60.91	76.29
	Wanda	64.15	73.34	60.33	77.16
	DSNoT	63.57	73.20	59.99	77.70
	OATS	**65.75**	**74.99**	**62.46**	**77.89**
50%	SparseGPT	53.22	67.63	53.60	72.47
	Wanda	54.57	69.76	49.83	72.04
	DSNoT	54.28	68.65	49.20	72.76
	OATS	**59.99**	**72.28**	**56.46**	**74.79**

Table 2: Comparison of average five-shot accuracies (%) on MMLU under different compression rates.

Compression	Method	Phi-3		Llama-3	
		Mini (3.8B)	Medium (14B)	8B	70B
0%	Dense	71.99	74.27	69.79	75.27
30%	SparseGPT	70.63	**74.53**	69.08	75.07
	Wanda	70.66	74.05	68.63	75.19
	DSNoT	71.20	74.03	68.98	**75.54**
	OATS	**71.48**	74.04	**69.34**	75.24
40%	SparseGPT	69.18	74.40	67.58	74.63
	Wanda	68.80	73.01	67.04	74.10
	DSNoT	69.08	72.90	66.65	74.29
	OATS	**70.04**	**74.46**	**68.68**	**74.88**
50%	SparseGPT	66.36	73.25	64.66	73.17
	Wanda	65.03	70.96	63.27	72.85
	DSNoT	65.33	71.12	62.74	72.91
	OATS	**68.41**	**73.39**	**65.71**	**73.30**

Table 3: Comparison of average zero-shot accuracies (%) under different compression rates. Task-specific scores can be found in Appendix A.12.

Compression	Method	Phi-3		Llama-3	
		Mini (3.8B)	Medium (14B)	8B	70B
0%	Dense	9.50	6.21	10.17	2.68
30%	SparseGPT	11.19	7.48	9.71	3.24
	Wanda	10.71	7.28	9.39	3.28
	DSNoT	10.51	7.11	**9.36**	3.27
	OATS	**10.27**	**6.85**	9.59	**3.07**
40%	SparseGPT	13.03	8.52	10.01	3.99
	Wanda	12.59	8.49	9.74	4.08
	DSNoT	12.17	8.24	9.60	4.10
	OATS	**11.53**	**7.70**	**9.24**	**3.68**
50%	SparseGPT	16.80	9.89	11.95	5.27
	Wanda	17.23	10.12	12.36	5.38
	DSNoT	16.68	9.96	12.41	5.58
	OATS	**15.18**	**9.05**	**10.87**	**4.78**

Table 4: Comparison of perplexity (lower is better) on WikiText-2 under different compression rates.

***Tables 2, 3, 4**: Comparative results for five-shot, zero-shot accuracy, and perplexity.*

- On downstream tasks (e.g., MMLU, ImageNet):
    - OATS consistently maintains performance better than traditional pruning approaches.

Compression	Method	ViT-Base	DinoV2-Giant
0%	Dense	80.33	86.55
30%	SparseGPT	80.21	86.46
	Wanda	**80.28**	86.47
	DSNoT	80.16	86.46
	OATS	80.15	**86.52**
40%	SparseGPT	79.58	86.39
	Wanda	79.34	86.32
	DSNoT	79.46	86.37
	OATS	**79.86**	**86.46**
50%	SparseGPT	78.44	86.04
	Wanda	76.19	85.81
	DSNoT	76.90	85.93
	OATS	**78.77**	**86.14**

Table 8: ImageNet validation accuracy (%).

Table 8 *here: Validation accuracy on ImageNet.*

Design Insights

Key innovations in OATS include:

- **Outlier Awareness:** Focuses compression on less critical weights while preserving influential outlier connections.
- **Joint Sparse + Low-Rank Strategy:** Improves retention of model capacity under heavy compression.
- **Flexible Ratio Control:** Allows fine-tuning the balance between sparse and low-rank terms, depending on target hardware constraints.
- **Low-Rank Decomposition Visualization:** Demonstrates that sparse and low-rank components capture complementary aspects of model behavior.

Figure 4 depicts the attention rollout for various images in the Microsoft COCO dataset (Lin et al., 2014) passed to a ViT-B that was compressed by 50%, with a rank ratio of 20%.

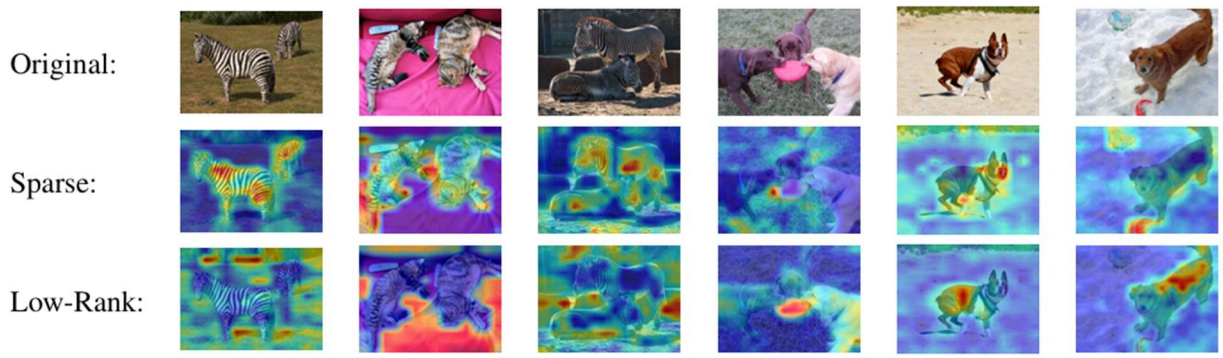


Figure 4: Attention rollout visualization applied to various images on the Microsoft COCO dataset.

***Figure 4**: Attention rollout visualizations comparing original, sparse, and low-rank reconstructions.*

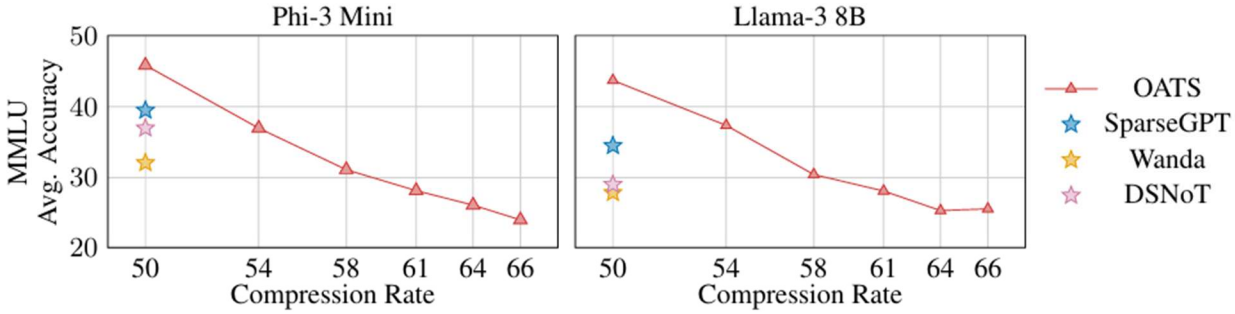


Figure 2: Experiments evaluating OATS with 2:8 structured sparsity on the sparse terms against 2:4 sparsity of state-of-the pruning algorithms. The rank ratio for OATS is varied to capture the performance across different compression rates.

***Figure 2** here: Structured sparsity experiments showing accuracy at different compression rates.*

Summary of Contributions

- **Introduces OATS**, a hybrid compression technique combining sparsity and low-rank modeling.
- **Maintains high accuracy** at aggressive compression levels across zero-shot, few-shot, and perplexity benchmarks.
- **Provides robust speedup**, achieving over 2× throughput improvements in practice.
- **Demonstrates versatility**, effective across multiple model scales (Phi-3, Llama-3, ViT).

- **Enables scalable, hardware-friendly compression** without needing retraining.

Why It Matters

OATS provides a **production-ready sparsification method** that delivers practical compression benefits without the retraining complexity typical of many pruning methods. By carefully modeling outlier activations, it enables deploying large models under tighter memory, computation, and latency budgets — critical for modern AI workloads across devices and cloud environments.

Where to Use

- **Deployment of LLMs and Vision Transformers** on resource-constrained servers and edge devices.
- **Data centers** optimizing inference cost and latency.
- **Applications demanding real-time response**, such as chatbots, recommendation systems, and image retrieval.

Section 3: Software and Compiler-Level Enhancements for Scaling Sparse Transformers in HPC

4.1 Paper: ByteTransformer: Compiler-Driven High-Performance for Variable-Length Inputs

ByteTransformer introduces a comprehensive set of **software and compiler optimizations** that eliminate padding overhead, fuse memory-bound operations, and rearchitect critical kernels to achieve high performance across variable-length inputs.

Unlike traditional frameworks that pad input sequences to a uniform maximum length (e.g., TensorFlow XLA, PyTorch JIT), ByteTransformer **removes padded computation entirely** via a **prefix-sum-based packing algorithm** and fuses multiple transformer operations into single kernels to minimize memory access and kernel launch overhead.

4.1.1 Padding-Free Tensor Transformation

To eliminate redundant computations on padded tokens, ByteTransformer introduces a **zero-padding removal algorithm**.
It applies **prefix sum** operations across the mask matrices to identify valid tokens, dynamically packing inputs for all transformer layers while maintaining semantic correctness.

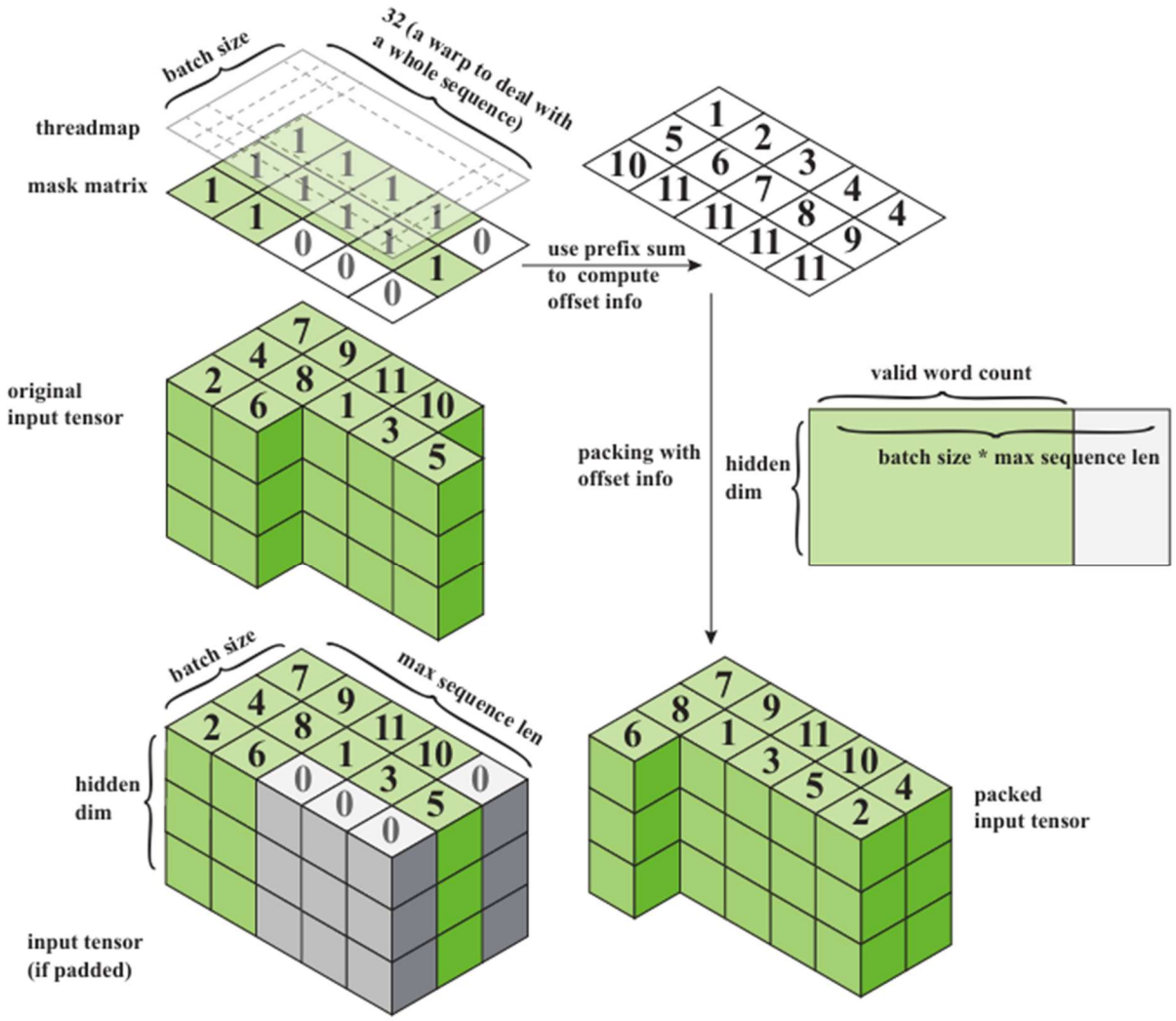


Fig. 4: The zero padding algorithm.

Figure 4 - Diagram of the zero-padding algorithm, showing prefix sum, packed tensors, and thread mapping.

This method reduces computation from batch_size × max_seq_len to valid_token_count, immediately lowering both FLOPs and memory usage across the pipeline.

4.1.2 Memory-Bound Kernel Fusion

Memory-bound operations like layer normalization, bias addition, and GELU activations are fused directly into surrounding compute kernels:

- **LayerNorm Fusion**: Fuses bias addition and layer normalization into a single memory access.
- **GELU Fusion**: Reuses GEMM outputs held in registers to apply bias and GELU activation before writing to global memory.

Performance Impact:

- LayerNorm fusion improved performance by **61%**.
- Bias and GELU fusion yielded an additional **24%** speedup.

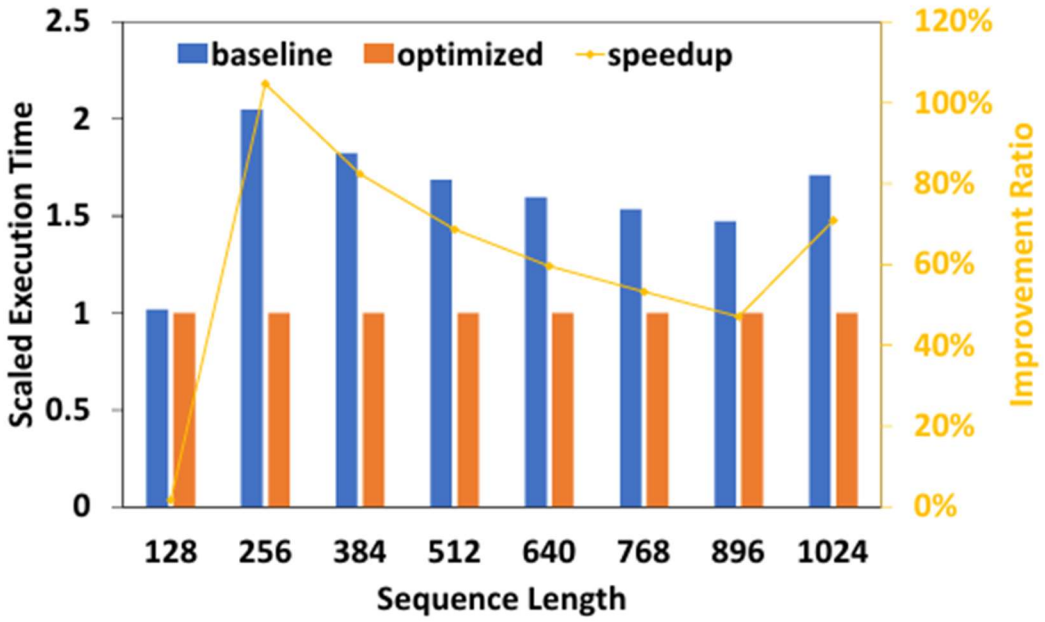


Fig. 9: Kernel fusion for add-bias and layernorm on a $(\texttt{batch\_size} \cdot \texttt{seq\_len}) \times \texttt{hidden\_dim}$ tensor. Here we profile for 16 batches with the hidden dimension fixed to 768 under the standard BERT configuration.

Figure 9 - Kernel fusion improvement for add-bias and layer normalization.

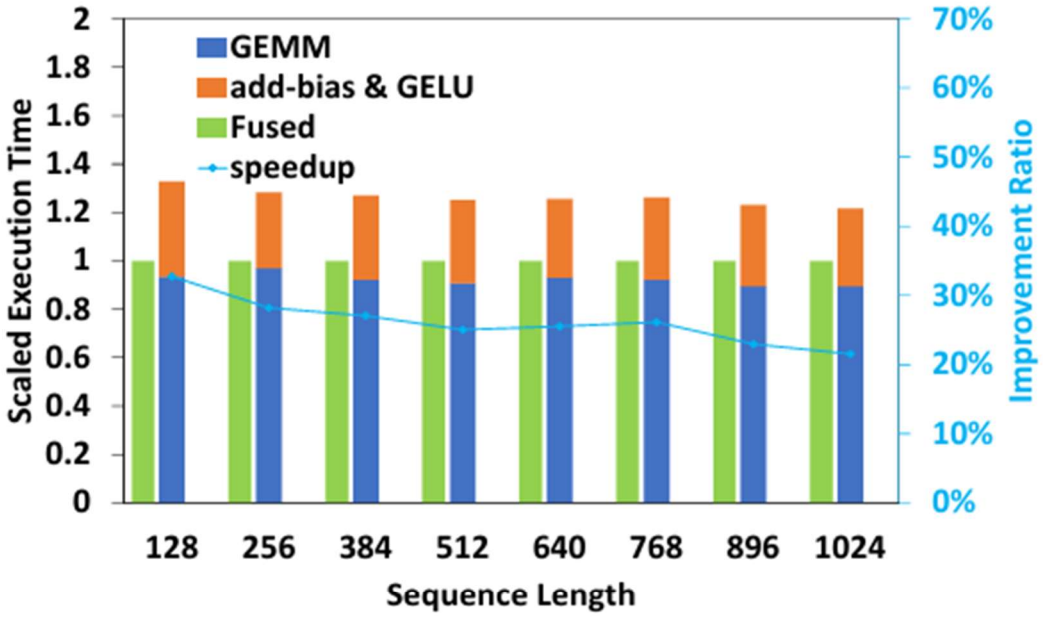


Fig. 10: Kernel fusion for GEMM, add-bias, and GELU. The shape of output tensor is $(\texttt{batch\_size} \cdot \texttt{seq\_len}) \times (\texttt{scale} \cdot \texttt{hidden\_dim})$. Here we profile for 16 batches with the hidden dimension and the scale factor fixed to 768 and 4 under the standard BERT configuration.

Figure 10: Kernel fusion improvement for GEMM + add-bias + GELU.

4.1.3 Fused Multi-Head Attention (MHA)

The Multi-Head Attention (MHA) module, traditionally composed of separate GEMMs and a softmax, is **fused into a single kernel** in ByteTransformer. Two strategies are employed:

- **Short Sequences**:
  For sequences under 384 tokens, **shared memory and register tiling** are used to hold intermediate matrices (Q, K, V) and compute attention without redundant loads.
- **Long Sequences**:
  For longer sequences, a **Grouped GEMM** technique is applied, allowing multiple attention problems of varying shapes to be processed efficiently in a round-robin scheduler.

Figure Placement:

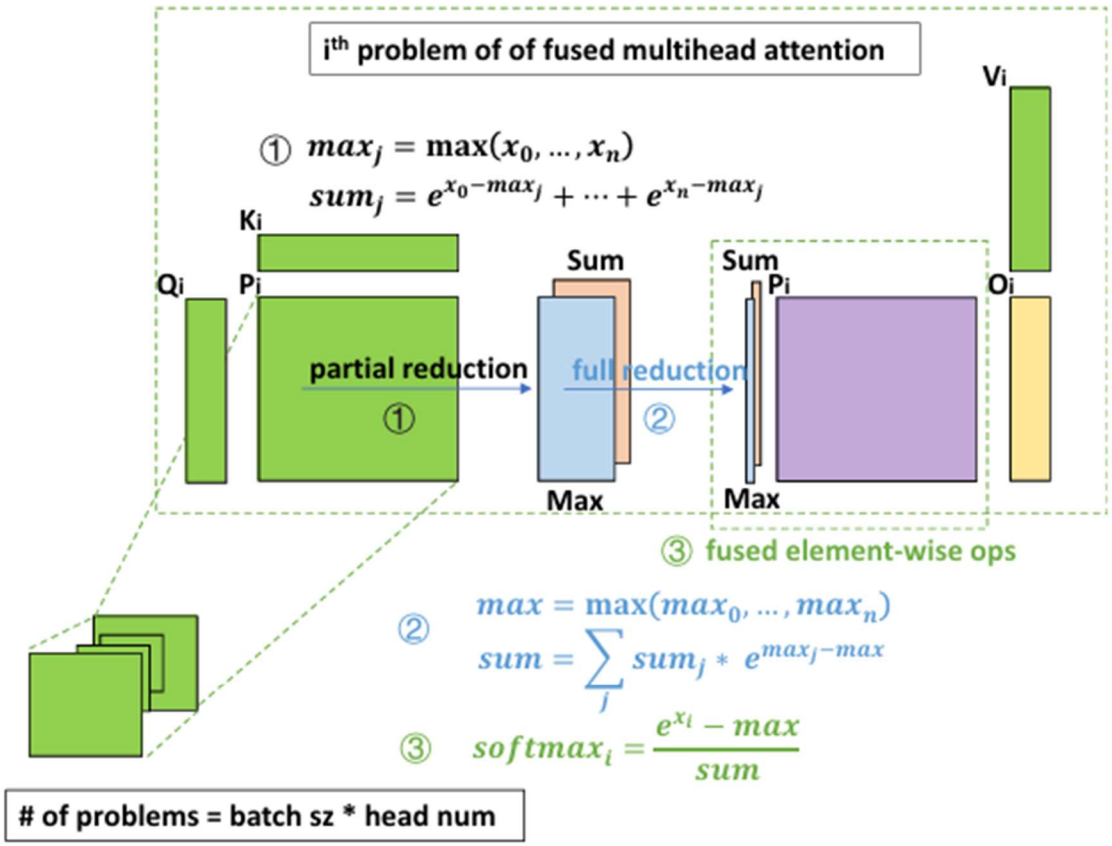


Fig. 6: Grouped-GEMM-based FMHA. The prototype of our fused MHA has been upstreamed to and released with CUTLASS 2.10. Source codes are available at [32].

Figure 6: Grouped-GEMM-based fused MHA illustration.

Key compiler-level techniques:

- Softmax computation is fused into the GEMM epilogue to hide memory latency.
- Elementwise normalization and reduction operations are fused directly into matrix multiplication loops.
- Warp-wide prefetching optimizes scheduler overhead.

Performance Gains:

- The fused MHA outperforms standard PyTorch MHA by **6.13×**.
- Overall, MHA fusion contributes a **19%** speedup in end-to-end transformer performance compared to standard implementations.

4.1.4 End-to-End Compiler Stack Acceleration

By systematically applying these compiler-driven optimizations:

- Kernel launch overhead is minimized.
- Tensor global memory traffic is reduced.
- Memory latency is hidden behind GEMM operations.
- Redundant calculations on padded tokens are fully eliminated for the entire forward pass.

Cumulative Impact (Figure Placement):

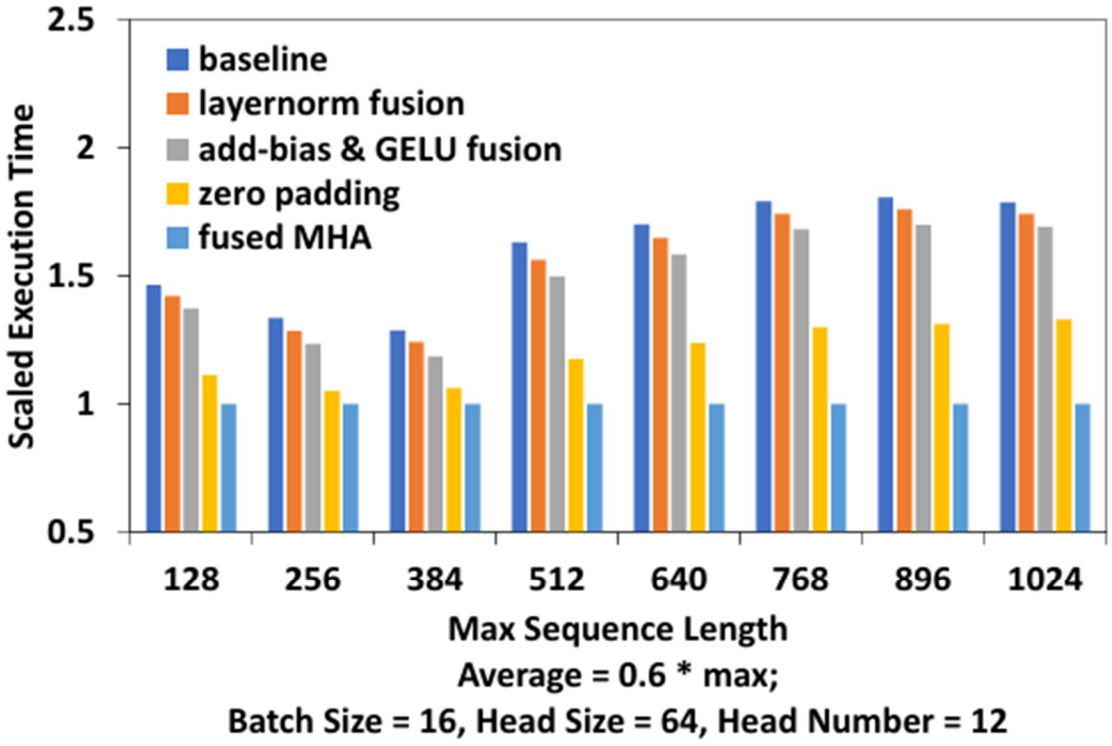


Fig. 14: Single-layer BERT transformer with step-wise optimizations. Each variant includes all previous optimizations.

Figure 14: Stepwise optimization improvements on single-layer BERT.

ByteTransformer achieves a **60% total speedup** compared to the baseline padded implementation.

Why it matters:

ByteTransformer offers a practical, compiler-optimized solution to transformer inference overhead by eliminating padding, fusing memory-bound ops, and optimizing MHA execution. It directly improves throughput in GPU inference pipelines with variable-length inputs."

Where to use:

ByteTransformer is best applied to LLM inference serving systems, real-time NLP deployments, and any environments with dynamic sequence lengths such as document classification, summarization, and dialogue modeling.

4.2 Paper: Understanding and Mitigating Spurious Correlations in Text Classification with Neighborhood Analysis

Pretrained language models (PLMs) often overfit to spurious correlations during fine-tuning, resulting in reduced generalization and poor robustness to distribution shifts. This paper proposes a **Neighborhood Analysis** framework to detect how token-level representation drift during fine-tuning leads to spurious behaviors, and introduces the **NFL (doN't Forget your Language)** regularization family to mitigate these effects without requiring auxiliary data.

Method Overview

Neighborhood Analysis monitors the **semantic structure of token embeddings** before and after fine-tuning. By quantifying changes in neighborhood relations, the method detects when tokens that should remain unrelated become spuriously clustered.

To address these shifts, the paper proposes multiple **NFL variants**:

- **NFL-F**: Freezes all pretrained parameters, training only the classification head.
- **NFL-CO**: Constrains the output token representations to remain close to pre-finetuning embeddings.
- **NFL-CP**: Constrains parameter drift via L2 regularization from pretrained weights.
- **NFL-PT**: Applies prompt-tuning, training only a small prompt embedding while freezing the PLM.

Implementation and Algorithmic Details

Neighborhood Analysis is performed using cosine similarity between token embeddings extracted from PLMs such as RoBERTa and BERT. Drift is measured as the change in nearest-neighbor relationships before and after fine-tuning.

NFL regularizations are integrated into standard cross-entropy training objectives:

- For NFL-CO, an auxiliary loss term penalizes token embedding shifts.
- For NFL-CP, an auxiliary loss term penalizes parameter updates relative to initialization.
- NFL-F and NFL-PT structurally prevent drift by freezing parameters.

All methods are implemented using Huggingface Transformers and standard PyTorch optimizers.

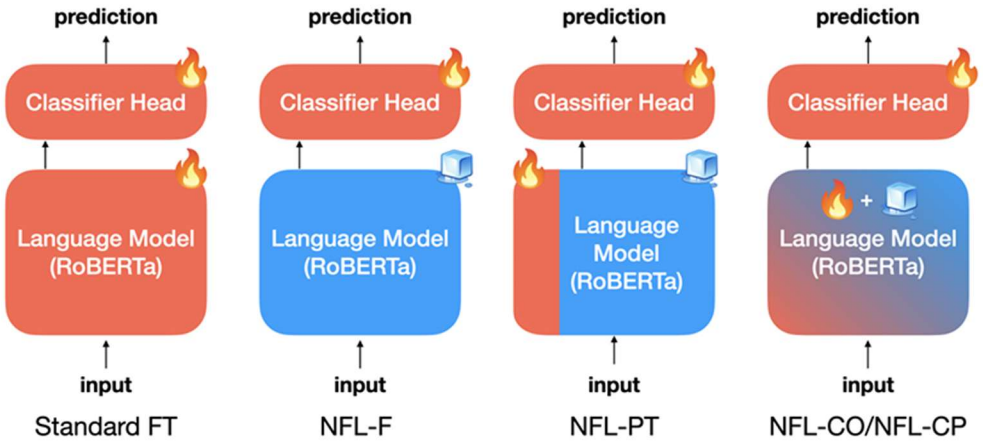


Figure 2: Comparison of fine-tuning and NFL. Red and blue regions represent trainable and frozen parameters respectively. Standard fine-tuning: every parameter is trainable; NFL-F: only the classification head is trainable; NFL-PT: the continuous prompts and the classification head are trainable; NFL-CO/NFL-CP: every parameter is trainable but changes in the language model are restricted by the regularization term in the loss function.

Figure 2 - Visualization of NFL regularization structures.

Efficiency and Hardware Scaling

NFL variants reduce training compute and memory demands:

- **NFL-F** and **NFL-PT** avoid updating large model parameters, decreasing GPU memory use.
- By preserving pretrained knowledge, they avoid extensive retraining or reliance on external debiasing datasets.

- NFL-PT is particularly efficient, requiring only small trainable prompts and achieving robustness with minimal parameter updates.

Accuracy and Generalization

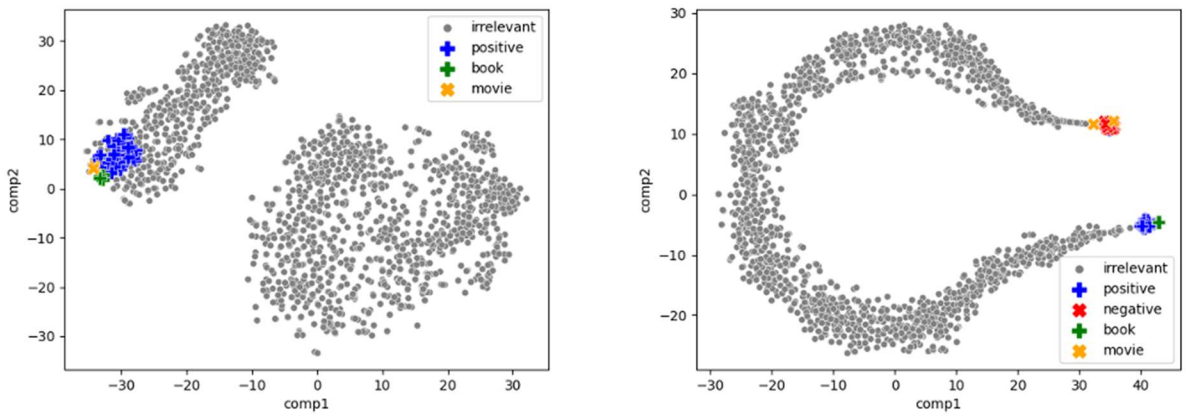
On Amazon sentiment and Jigsaw toxicity datasets:

- NFL variants substantially close the gap between biased and unbiased test sets.
- NFL-CP and NFL-CO achieve similar or superior robust accuracy compared to idealized models trained without spurious correlations.

	Amazon binary			Jigsaw		
Method	Biased acc	Robust acc	Δ	Biased acc	Robust acc	Δ
Trained solely on $\mathcal{D}_{\text{biased}}$						
RoBERTa	**95.7**	53.3	-42.4	**86.5**	50.3	-36.2
NFL-F	89.5	77.3	-12.2	75.3	70.3	-5.0
NFL-CO	92.9	85.7	-7.2	78.9	73.4	-5.5
NFL-CP	95.3	91.3	-4.0	84.8	**80.9**	**-3.9**
NFL-PT	94.2	**92.9**	**-1.3**	82.5	78.2	-4.3
Trained on $\mathcal{D}_{\text{unbiased}}$						
DFR (5%)	93.6	83.1	-9.5	86.3	75.0	-11.3
DFR (100%)	93.4	88.9	-4.5	85.9	78.0	-7.9
Ideal Model	94.8	95.6	0.8	85.2	82.2	-3.0

Table 5: Amazon binary and Jigsaw results. Robustness gap Δ is robust accuracy − biased accuracy. NFL exhibits low degradation when exposed to spurious correlation. Bold text represents the highest score among all models, with the exception of the scores obtained by the ideal model.

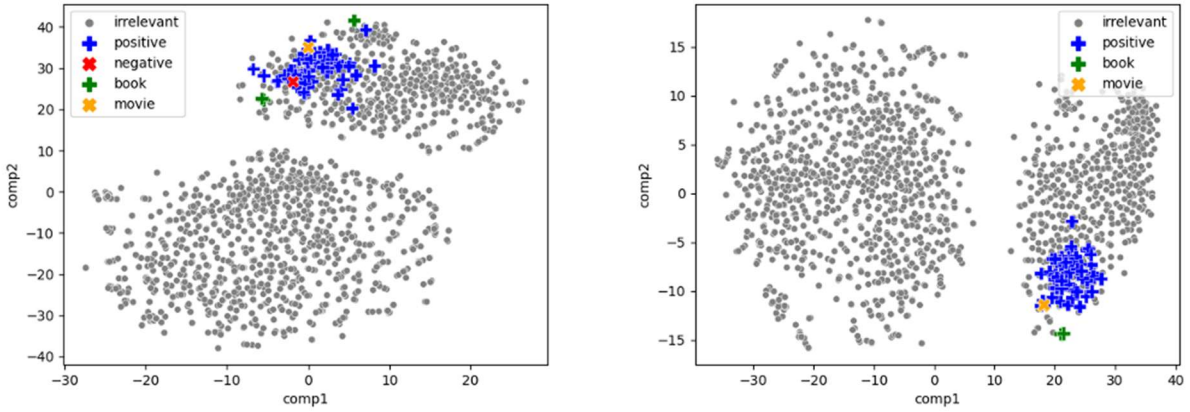
Table 5 demonstrates NFL's success in improving robust test performance.



(a) Initial (b) Standard fine-tuning

Figure 1: t-SNE projections of representations before and after fine-tuning. BOOK, MOVIE erroneously align with genuine positive, negative tokens respectively after fine-tuning, preventing the classifier from distinguishing between spurious and genuine tokens.

Figure 1: t-SNE visualization of spurious neighborhood collapse after standard fine-tuning.



(a) NFL-CO (b) NFL-CP

Figure 3: t-SNE projections of representations after fine-tuning with NFL-CO/NFL-CP. By preventing the formation of erroneous clusters, NFL learns robust representations.

Figure 3: Post-NFL t-SNE plots showing preserved token clusters.

Design Insights

- **Neighborhood Analysis** provides a lightweight, unsupervised method for diagnosing model brittleness.
- **Unified Regularization Framework**: NFL enables flexible trade-offs between model robustness, training cost, and deployment efficiency.
- **Prompt-tuning synergy**: NFL-PT shows that even lightweight prompt learning can maintain strong anti-spurious behaviors.

Summary of Contributions

- Introduces **Neighborhood Analysis** for diagnosing token-level drift during fine-tuning.
- Proposes the **NFL family** (Frozen, Constrained Output, Constrained Parameters, Prompt-Tuning) for spurious correlation mitigation.
- Demonstrates robust accuracy improvements across multiple spurious-sensitive benchmarks without needing auxiliary datasets.
- Offers scalable, hardware-efficient regularization methods suitable for large pretrained transformers.

Why it matters

NFL techniques enable **efficient, hardware-friendly fine-tuning** by reducing memory usage, compute demands, and dependence on additional debiasing datasets. Neighborhood Analysis further provides **an unsupervised method** to diagnose and mitigate model fragility at scale. This is essential for deploying robust language models in real-world settings without prohibitive retraining costs.

Where to use

NFL regularization is best suited for:

- **Text classification and toxicity detection** tasks vulnerable to spurious correlations.
- **Domain adaptation and low-resource settings** where retraining large PLMs is impractical.
- **Resource-constrained deployment pipelines** seeking robust models with minimal fine-tuning overhead.

Section 4: Hardware-Aware Sparse Computation Beyond Attention

While Section 4 focuses on attention sparsity as a direct solution to the $O(L^2)$ complexity in Transformer models, other components—such as feedforward networks (FFNs), expert routing, and kernel-level execution—also present substantial computational bottlenecks. The methods outlined in this section do not reduce the attention matrix size or complexity but instead introduce **structured sparsity and routing strategies** that **optimize performance in GPU/TPU-based and multi-node environments**. These approaches serve as **complementary mechanisms** that can be combined with attention sparsity for full-model acceleration.

5.1 Structured Weight Sparsity for GPU Efficiency

Paper: Beyond 2:4: Exploring V:N:M Sparsity for Efficient Transformer Inference on GPUs

Structured sparsity formats like 2:4 have been adopted in production settings due to their alignment with GPU tensor core execution, particularly on NVIDIA Ampere and Hopper architectures. However, 2:4 sparsity imposes fixed pruning constraints, which can limit compression and generalization.

To address this, Zhao et al. (2025) propose a generalization called V:N:M sparsity, introduced in *Beyond 2:4: Exploring V:N:M Sparsity for Efficient Transformer Inference on GPUs*. This approach decouples the 2:4 constraint and enables formats such as 64:2:8 or 128:2:6, which are still compatible with Sparse Tensor Cores (SpTC) while offering greater flexibility in pruning. Figure 1

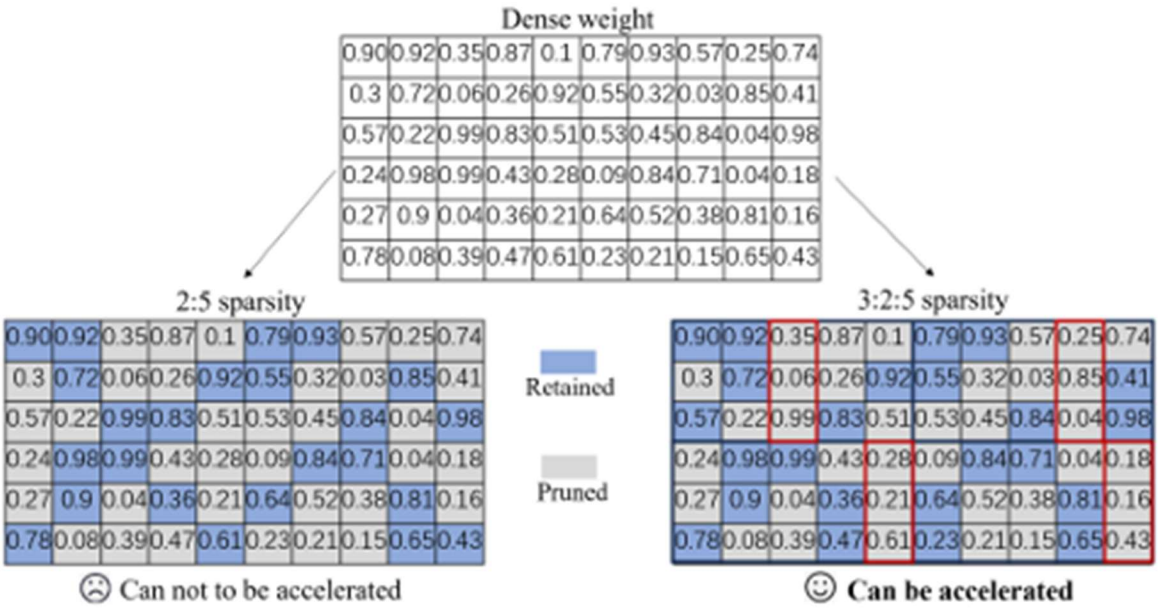


Figure 1: An example on N:M sparsity and V:N:M sparsity

Figure 1 (adapted from Zhao et al., 2025): Comparison of NVIDIA's 2:4 structured sparsity vs the generalized V:N:M format.

The core components of their method include:

- Compressed weight representations optimized for sparse matrix-multiply (SpMM) operations.
- A V:N:M-specific channel permutation (CP) mechanism to recover accuracy in highly pruned models.
- To enable deployment-ready sparse models, the authors design a three-stage LoRA-based fine-tuning routine for retraining with minimal memory overhead.

Their experiments show that fixed sparse masks (used in V:N:M) lead to more stable training compared to dynamic masking strategies. Figure 7 illustrates this result, showing that models trained with fixed masks achieve lower and more consistent loss than their dynamic counterparts. This supports the argument for V:N:M as not only an inference-efficient format, but also a training-stable one.

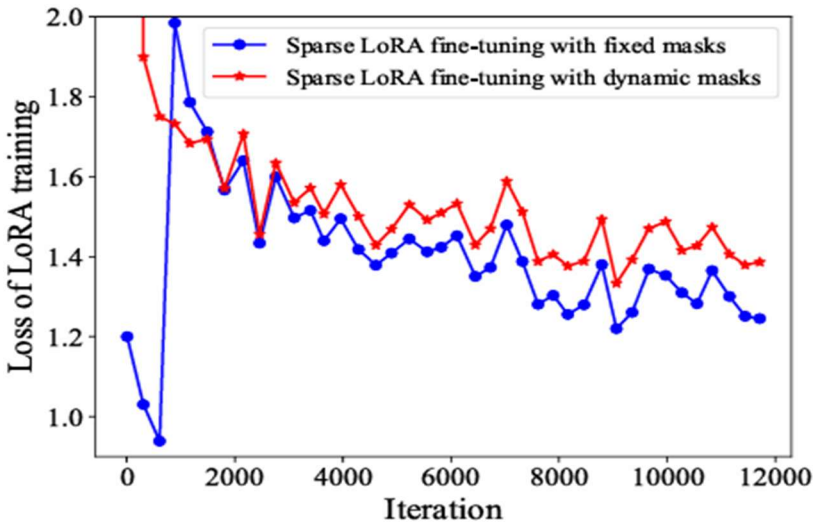


Figure 7: Loss of Llama2-7B during sparse LoRA fine-tuning with dynamic masks and fixed masks, respectively.

Figure 7 (adapted from Zhao et al., 2025): LoRA training loss curves comparing fixed versus dynamic sparsity mask strategies. Fixed V:N:M masking leads to more stable convergence.

This design enables high sparsity without retraining from scratch, achieving:

- 73.8% parameter reduction and 71.6% FLOP reduction
- 1.49× inference speedup on LLaMA2-7B
- 2.02× speedup on DeiT-base

All benchmarks are executed on NVIDIA A100 using sparse-aware SpMM kernels.

Although V:N:M sparsity is applied to MLP and projection layers—not attention matrices—it significantly reduces total compute and is highly complementary to sparse attention mechanisms. In throughput-critical environments, such as real-time inference on HPC infrastructure, this combined sparsity strategy is essential.

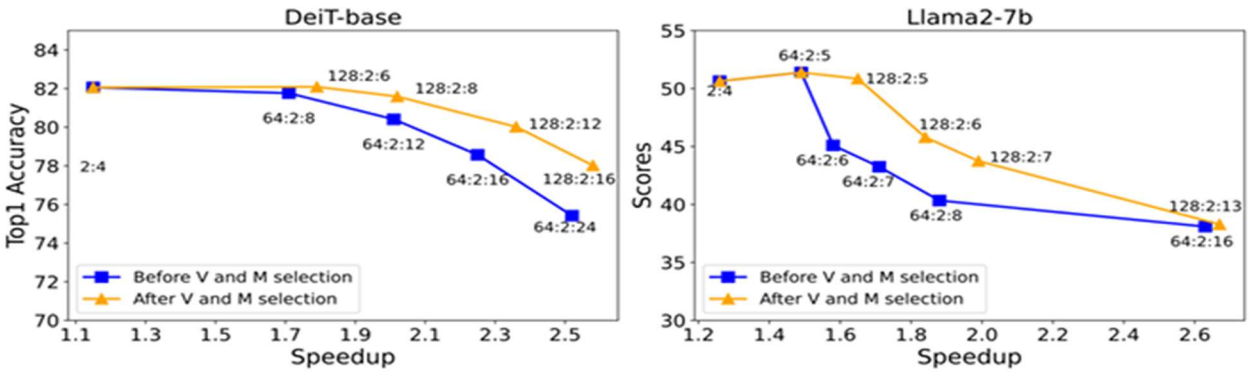


Figure 4: The speedup-accuracy curves of V:N:M-sparse Transformer. a) Top-1 accuracy of DeiT-base using TS2 with different V and M values. Accuracy of dense DeiT-base: 81.84%. b) Average scores of Llama2-7B using TS3 on 5-shot tasks with different V and M values. Average score of dense Llama2-7B: 61.99. Average score of 2:4-sparse Llama2-7B: 50.76. The speedup-accuracy curves using TS1 are shown in Figure 10 in Appendix F. More results for larger Transformers are shown in Figure 11, 12 and 13 in Appendix G, respectively

Figures 4a and 4b plot the speed-accuracy tradeoffs for LLaMA2-7B and DeiT-base under various configurations, demonstrating the performance gains of the V:N:M format.

This method provides a strong example of how structured pruning can be aligned directly with low-level hardware optimizations to yield practical gains in latency and memory efficiency.

- Proposes a generalization of NVIDIA's 2:4 structured sparsity to **V:N:M sparsity**, enabling >50% pruning with compatibility for **sparse tensor cores**.
- Supports arbitrary sparsity formats (e.g., 64:2:8) with compressed weight representations optimized for **sparse matrix-multiply (SpMM) kernels**.
- Introduces **V:N:M-specific channel permutation (CP)** to recover accuracy under aggressive sparsity.
- Implements a **three-stage LoRA-based training routine** to adapt sparse patterns using low memory.
- Achieves:
    - **73.8% parameter reduction** and **71.6% FLOP reduction**.
    - **1.49× speedup** on LLaMA2-7B and **2.02×** on DeiT-base using sparse-aware kernels on NVIDIA Ampere+ GPUs.

Why it matters:
V:N:M sparsity is not applied to attention matrices, but it significantly reduces MLP and linear layer compute. This type of structured sparsity aligns with current and emerging GPU

hardware features, making it especially relevant for **real-time inference** and **throughput-critical applications** in HPC pipelines.

Where to use:
V:N:M can be layered with attention sparsity methods to achieve **full-model compression**, especially for deployment on hardware like NVIDIA A100/H100 or custom inference accelerators with SpTC support.

5.2 Sparse Expert Routing and MoE Scaling

Paper: PanGu-Σ: Towards Trillion-Parameter Language Model with Sparse Heterogeneous Computing

Efficient scaling of sparse Transformer models requires not only reducing computation per layer, but also minimizing routing overhead during expert selection — especially in multi-node or heterogeneous environments. Traditional Mixture-of-Experts (MoE) architectures often rely on top-k gating, which incurs memory and latency penalties due to global communication and softmax operations.

To address this, **PanGu-Σ (Zeng et al., 2025)** introduces **Random Routed Experts (RRE)**, a sparse expert routing strategy that avoids learnable gates altogether. Instead, tokens are first classified into a domain (e.g., code, English, Chinese) and then randomly routed to one of the experts associated with that domain. This reduces memory usage, eliminates gate overhead, and enables hardware-friendly expert activation. Only **10% of experts are active per forward pass**, making it highly scalable across multi-GPU or NPU-CPU hybrid systems.

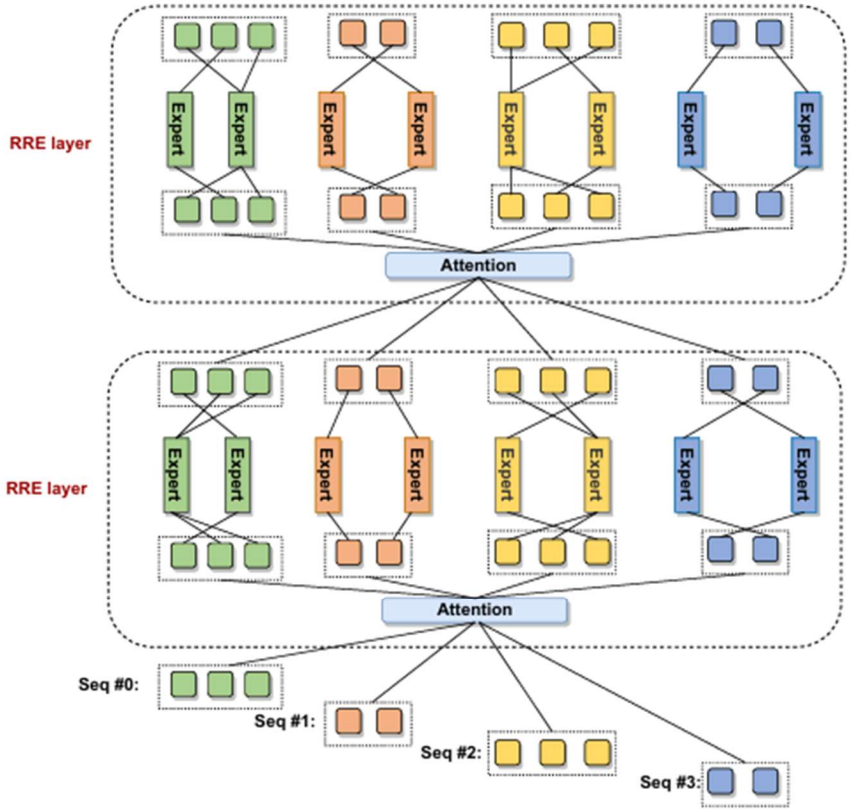


Figure 2: Random Routed Experts (RRE) in PanGu-Σ . The token is first routed to a group of experts by domain, and then randomly routed to one of the experts in that domain. There is no learnable routers in the model.

Figure 2 (adapted from PanGu-Σ, 2025): Architecture of the RRE routing mechanism. Tokens are assigned to domain-specific expert groups and routed randomly within the group, reducing the cost of top-k gating while maintaining expert specialization.

PanGu-Σ also introduces **Expert Computation and Storage Separation (ECSS)**, which decouples the execution of dense and sparse components across heterogeneous compute resources — such as **Ascend NPUs** and **KunPeng CPUs**. This enables more flexible scheduling in distributed training environments.

Key components of their approach include:

- **Random Routed Experts (RRE)**: Token-level sparse routing without learnable gates or softmax layers
- **Expert Computation and Storage Separation (ECSS)**: Enables expert-specific compute over heterogeneous hardware nodes
- **Sub-model extraction**: Allows partial model deployment by selecting a subset of domain-aligned experts without retraining

This architecture achieves strong performance while remaining hardware-efficient:

- **Activates only 10% of experts** per inference step
- Minimizes expert communication latency in multi-node setups
- Supports modular deployment in **real-time inference scenarios**

These innovations make PanGu-Σ a highly relevant solution for **sparse expert execution in HPC environments**, especially where energy, memory, or multi-architecture compatibility are key constraints.

In addition to optimizing expert routing, PanGu-Σ also highlights practical training inefficiencies encountered during large-scale model deployment. Figures 6 and 7 from the original paper show the contrast between input formatting during pretraining versus fine-tuning.

During pretraining, sequences are packed into fixed-length blocks (e.g., 512 tokens), maximizing batch uniformity and minimizing padding (Figure 6). However, fine-tuning involves domain-specific tasks with variable-length inputs. As shown in Figure 7, this leads to excessive padding and poor memory efficiency — wasting GPU capacity and degrading throughput.

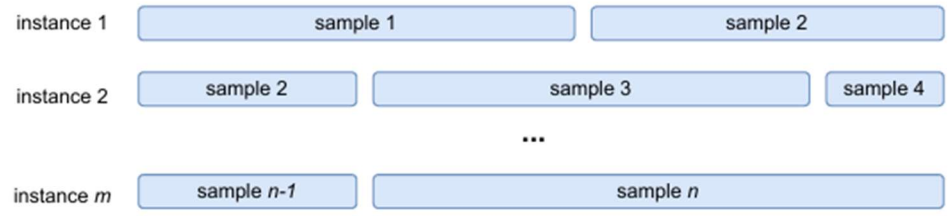


Figure 6: Input format during model pre-training.

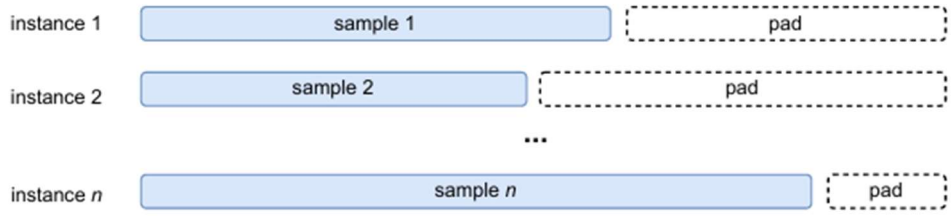


Figure 7: Input format during model fine-tuning.

Figure 6 (adapted from PanGu-Σ, 2025): Input packing strategy used during pretraining. Uniform sequence length enables high batch utilization.

Figure 7 (adapted from PanGu-Σ, 2025): Padding inefficiencies in fine-tuning with variable-length inputs lead to memory waste and reduced throughput.

This highlights a key lesson for HPML workflows: **compression and routing gains must be complemented by data-level batching strategies** to maintain efficiency across the entire training pipeline.

Paper: Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity

While attention sparsity has received widespread focus, the **Mixture-of-Experts (MoE)** paradigm also poses significant hardware challenges due to redundant expert activation and memory-intensive routing. **Wu et al. (2025)** propose **Samoyeds**, a hardware-optimized system that accelerates MoE-based LLMs by jointly leveraging structured sparsity in **both** model weights and **activations**. Introduced in *Samoyeds: Accelerating MoE Models with Structured Sparsity Leveraging Sparse Tensor Cores*, the system targets NVIDIA Sparse Tensor Core (SpTC)-based GPUs to deliver high throughput under practical MoE workloads.

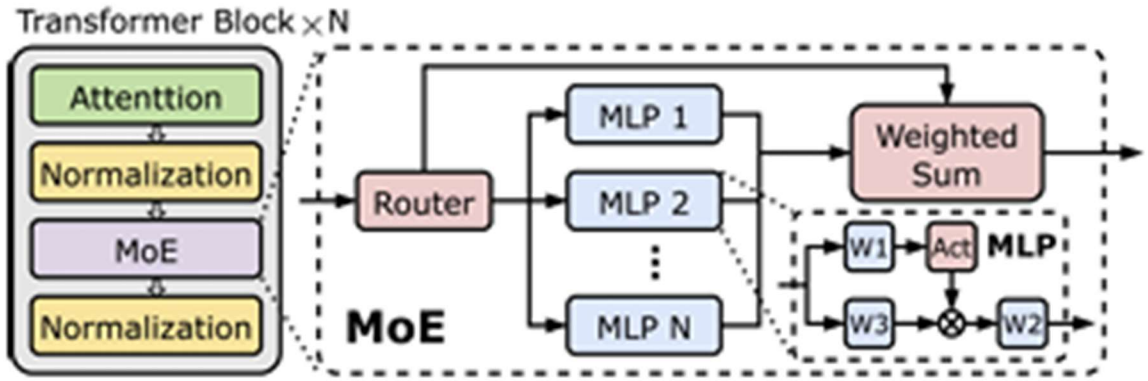


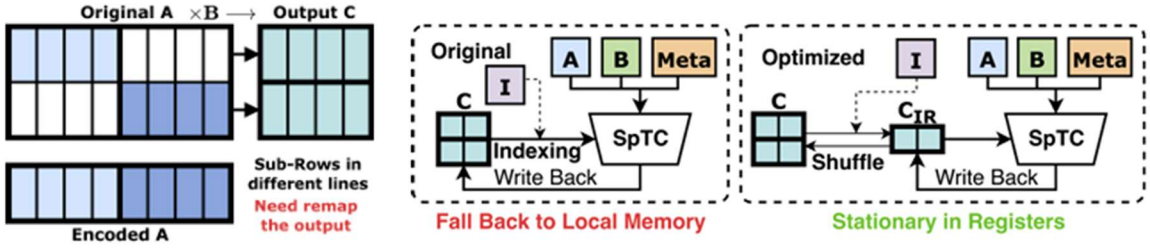
Figure 1. MoE LLM Architecture.

Figure 1 from the paper illustrates the architecture of a sparse MoE block, showing how tokens are routed through selected MLP experts via a sparse dataflow.

Key Contributions:

- **Dual-side structured sparsity** format for MoE layers:
    - **Weights:** Apply 2:4 and vector-wise sparsity compatible with SpTC.
    - **Activations:** Encode input sparsity induced by token routing into a “selection array.”

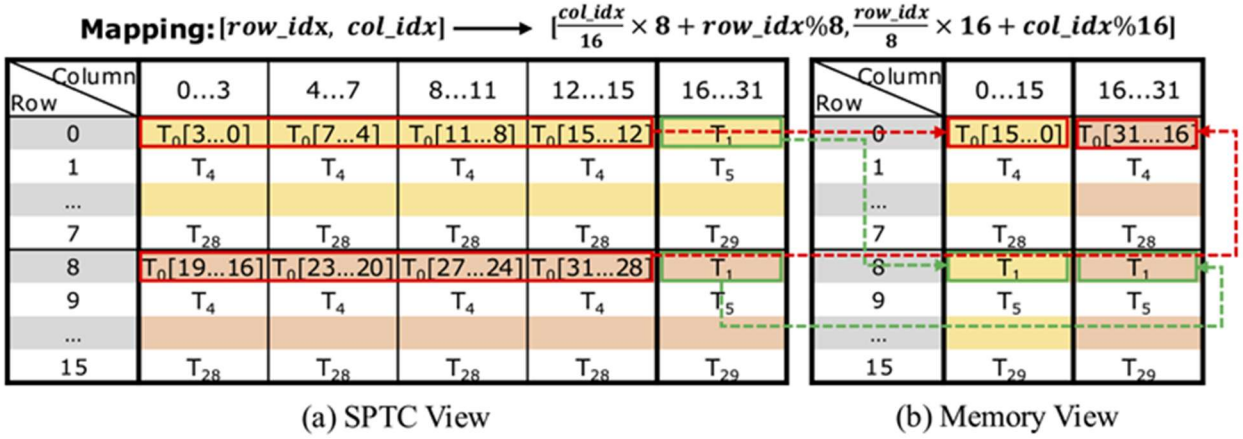
- **Custom sparse-sparse SpMM kernel** that respects SpTC instruction constraints and handles both structured sparse weights and activations.
- **System-level optimizations**, including:
    - 3-stage GPU memory **tiling**



(a) Problem Illustration. (b) Introducing Intermediate Registers.

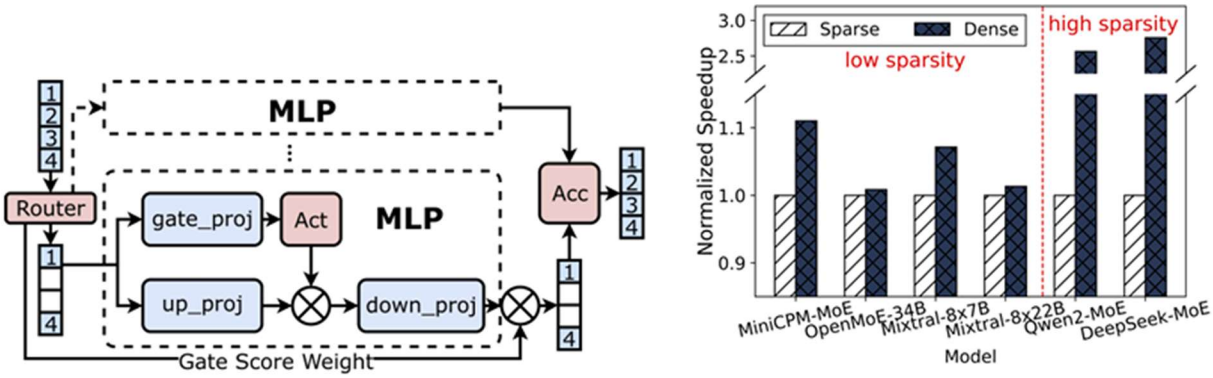
Figure 9. Data Stationary Optimization for Output Matrix.

- **Data stationary reuse** using intermediate register remapping (see **Figure 9**)



(a) SPTC View (b) Memory View

Figure 10. Packing Strategy with Reorganized Metadata.



(a) Output Sparsity in MoE Layer. (b) Kernel Performance Improvement Optimized Output Layout.

Figure 11. Layout Optimization for Kernel Output.

- Optimized metadata packing (see **Figure 10**) and output layout (see **Figure 11**) for reduced I/O overhead.

Performance:

- **Up to 1.99× kernel speedup** over VENOM (structured SpMM baseline).
- **2.36× model-level speedup** over Transformers.
- **4.41× increase in max batch size**, enabling more efficient throughput at scale.
- **33× speedup over unstructured sparse baseline (Sputnik)** in some cases.

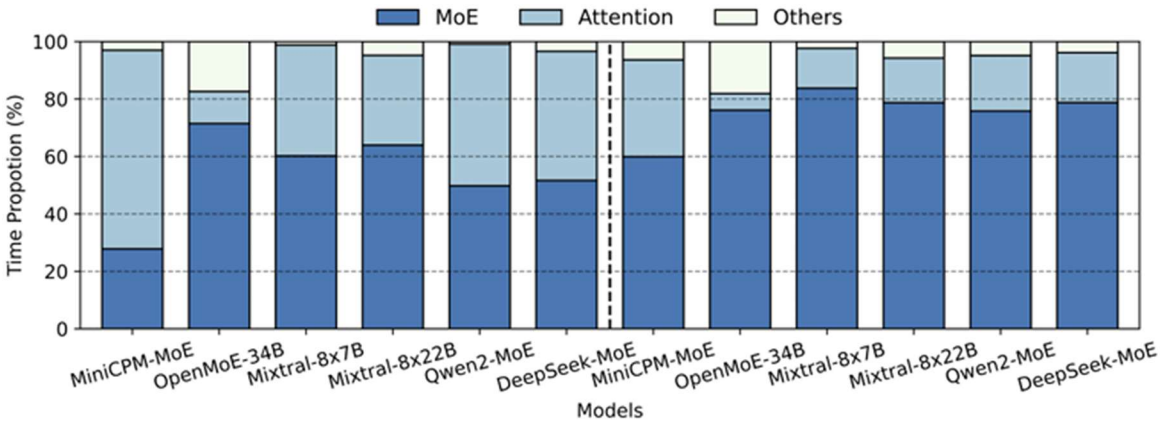


Figure 2. Time Breakdown of MoE Models. Left: Without Flash-Attention; Right: With Flash-Attention.

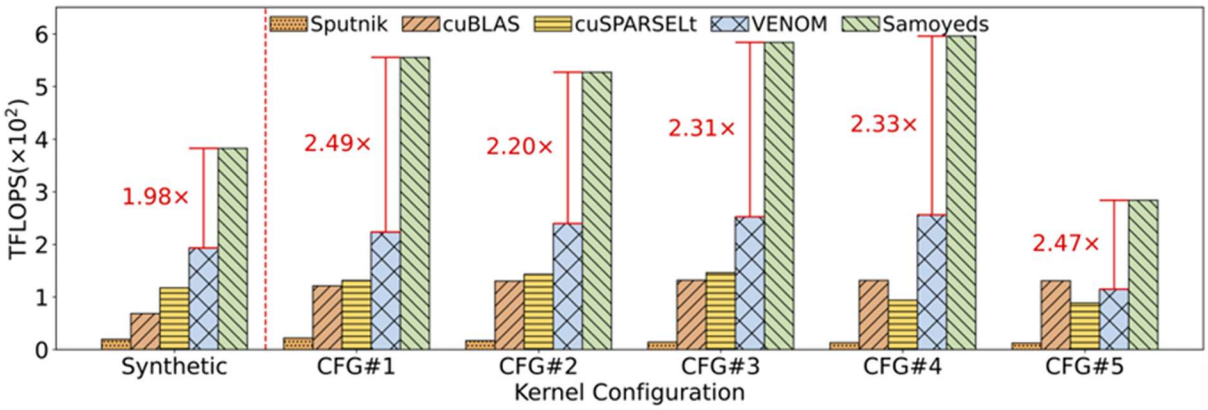


Figure 12. Kernel Performance Comparison on Synthetic and Realistic Benchmarks. The synthetic benchmark covers 238 sizes; the realistic benchmark reflects typical model sizes.

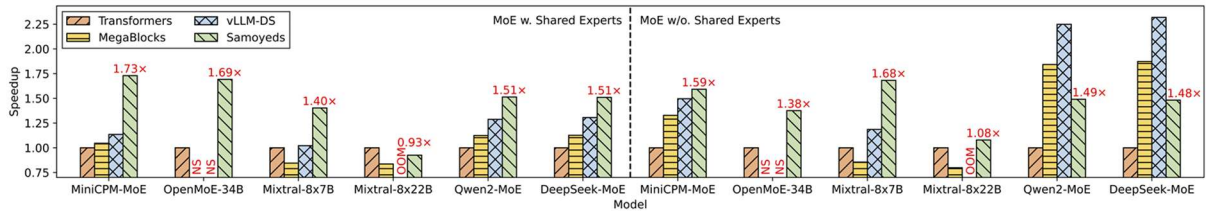


Figure 14. Execution Speedup for the MoE Layer. *NS* indicates *not supported* due to kernel incompatibility with OpenMoE-34B. *OOM* denotes out-of-memory errors preventing execution completion.

Figures 2, **12**, and **14** show runtime breakdowns of MoE bottlenecks, kernel performance comparisons, and layer-level speedup over other frameworks, respectively.

Why It Matters:

This work extends sparse efficiency beyond attention and MLPs, directly tackling the MoE execution bottleneck found in trillion-parameter LLMs. It enables real-time inference under tight memory and latency constraints on HPC hardware.

Where to use:
Ideal for organizations deploying massive LLMs where compute budget, throughput, and routing overhead are critical bottlenecks. These models can also be paired with sparse attention modules like MoA or SemSA for end-to-end gains.

5.3 MoE Kernel Optimization for Sparse Hardware

Paper: Samoyeds: Accelerating MoE Models with Structured Sparsity Leveraging Sparse Tensor Cores

While traditional sparse attention aims to reduce the quadratic $O(L^2)$ bottleneck of self-attention, **Switch Transformers (Fedus et al., 2022)** take a different route — they sparsify **feedforward computation** by activating a single expert per token, achieving scalable throughput for trillion-parameter models under constant compute budgets.

Their core contribution is **simplifying Mixture-of-Experts (MoE)** routing:

- Route each token to **only one expert** (k=1), unlike MoE's top-k.
- Use a lightweight, differentiable router that eliminates expensive softmax computation across many experts.

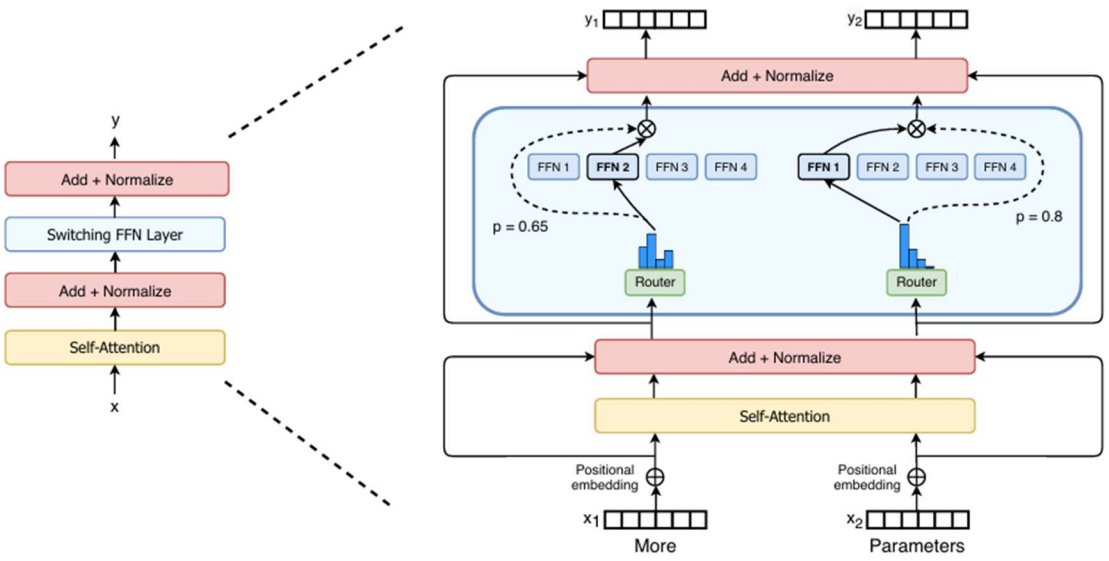


Figure 2: Illustration of a Switch Transformer encoder block. We replace the dense feed forward network (FFN) layer present in the Transformer with a sparse Switch FFN layer (light blue). The layer operates independently on the tokens in the sequence. We diagram two tokens (x_1 = "More" and x_2 = "Parameters" below) being routed (solid lines) across four FFN experts, where the router independently routes each token. The switch FFN layer returns the output of the selected FFN multiplied by the router gate value (dotted-line).

Figure 2 illustrates this idea: each token is routed independently to one expert via a simple gating mechanism, drastically reducing compute and memory usage.

Routing simplification leads to:

- **7× faster training** over T5-Base for the same compute budget.
- **2.5× speedup** compared to T5-Large even though T5-Large uses 3.5× more FLOPs per token.

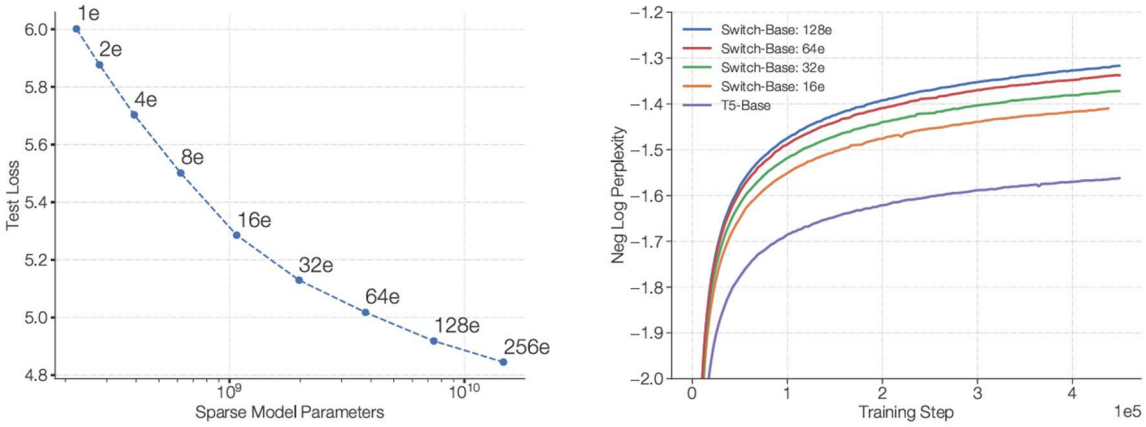


Figure 1: Scaling and sample efficiency of Switch Transformers. Left Plot: Scaling properties for increasingly sparse (more experts) Switch Transformers. Right Plot: Negative log perplexity comparing Switch Transformers to T5 (Raffel et al., 2019) models using the same compute budget.

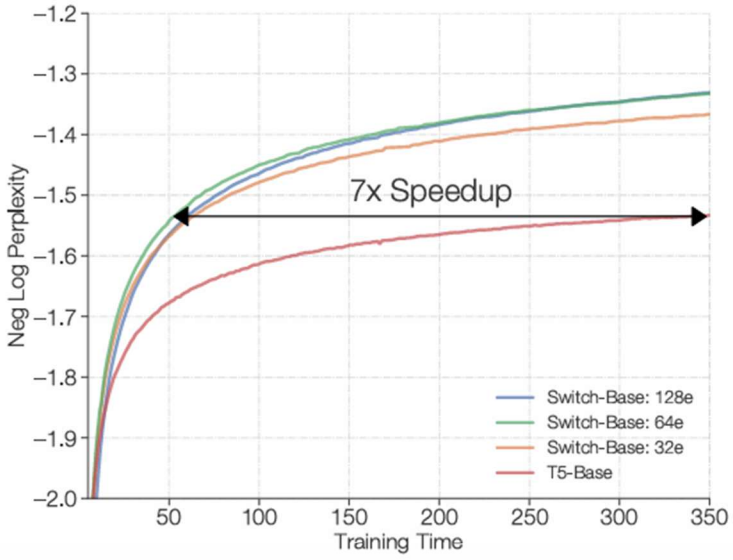


Figure 5: Speed advantage of Switch Transformer. All models trained on 32 TPUv3 cores with equal FLOPs per example. For a fixed amount of computation and training time, Switch Transformers significantly outperform the dense Transformer baseline. Our 64 expert Switch-Base model achieves the same quality in *one-seventh* the time of the T5-Base and continues to improve.

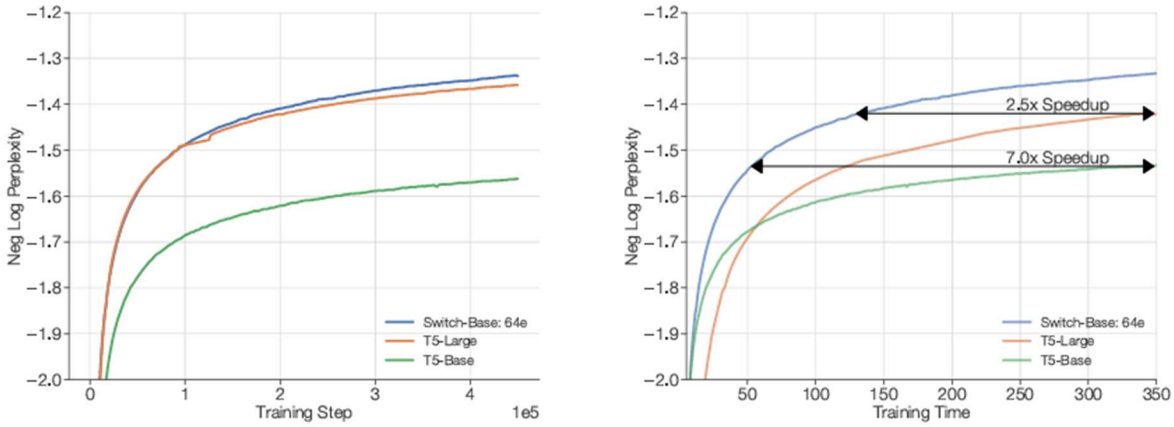


Figure 6: Scaling Transformer models with Switch layers or with standard dense model scaling. Left Plot: Switch-Base is more sample efficient than both the T5-Base, and T5-Large variant, which applies 3.5x more FLOPS per token. Right Plot: As before, on a wall-clock basis, we find that Switch-Base is still faster, and yields a 2.5x speedup over T5-Large.

Figures 1, 5, and 6 show performance trends: Switch Transformers improve perplexity, sample efficiency, and real-world time-to-train under constant hardware.

Switch Transformers also leverage a **highly efficient partitioning strategy**:

- **Expert Parallelism**: Each core processes a different expert.
- **Data & Model Parallelism**: Combined to balance FLOPs, memory, and communication costs.

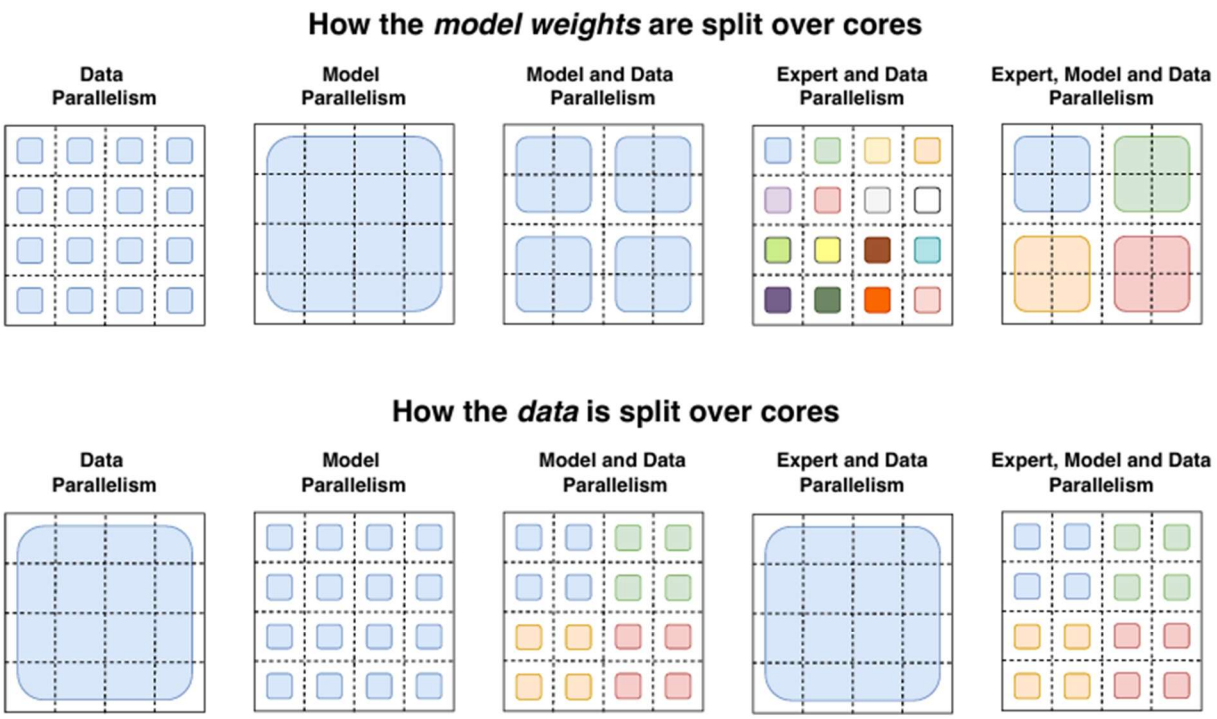


Figure 9: Data and weight partitioning strategies. Each 4×4 dotted-line grid represents 16 cores and the shaded squares are the data contained on that core (either model weights or batch of tokens). We illustrate both how the model weights and the data tensors are split for each strategy. **First Row:** illustration of how *model weights* are split across the cores. Shapes of different sizes in this row represent larger weight matrices in the Feed Forward Network (FFN) layers (e.g larger d_{ff} sizes). Each color of the shaded squares identifies a unique weight matrix. The number of parameters *per core* is fixed, but larger weight matrices will apply more computation to each token. **Second Row:** illustration of how the *data batch* is split across cores. Each core holds the same number of tokens which maintains a fixed memory usage across all strategies. The partitioning strategies have different properties of allowing each core to either have the same tokens or different tokens across cores, which is what the different colors symbolize.

Figure 9 visualizes how Switch models partition computation across cores using Mesh-TensorFlow's logical core mesh. This is essential for HPC deployments.

In terms of HPML relevance:

- They trade $O(L^2)$ attention compute for **token-level expert sparsity**, reducing memory pressure and enabling **multi-node training of massive models**.
- Their routing and communication optimization aligns with **HPC infrastructure constraints** (e.g., limited memory bandwidth, distributed load balancing).

This makes the Switch Transformer not just a scalable model, but a foundational pattern for **large sparse model design in high-performance ML pipelines**.

- Applies **dual-side structured sparsity** (in both weights and activations) to improve MoE layer execution efficiency.
- Develops a custom **sparse-sparse kernel** for NVIDIA GPUs using **mma.sp** instructions (Sparse Tensor Core support).
- Implements advanced scheduling: **tiling, register reuse, and fusion optimizations**.
- Benchmarks:
    - **18.76× kernel-level speedup** over Sputnik.
    - **1.99× model-level speedup** over VENOM.
    - **4.41× increase in max batch size** supported during inference.

Why it matters:
Samoyeds bridges the software-hardware gap by exploiting **structured GPU sparsity primitives** to run MoE-based Transformers faster and with lower memory. This is especially important for **real-time inference in resource-constrained or latency-sensitive HPC systems**.

Where to use:
Best suited for **NVIDIA Ampere+, Hopper, and Ada GPUs** with Sparse Tensor Core support. Can be integrated into MoE models using Switch or PanGu-style routing.

Applications in HPC

Sparse Transformer (OpenAI, 2019) — "Generating Long Sequences with Sparse Transformers"

Overview

The paper "Generating Long Sequences with Sparse Transformers" addresses the limitations of dense self-attention mechanisms, which impose quadratic memory and compute costs, thereby restricting the maximum sequence lengths that can be processed by Transformer architectures. By introducing a structured sparsity pattern into the attention matrix, OpenAI's Sparse Transformer model reduces the computational complexity from $O(n^2)$ to $O(n\sqrt{n})$, where n is the sequence length. This sparsity enables scaling to

significantly longer sequences while maintaining model performance across diverse generative tasks such as text, images, and audio.

Introduction

Dense self-attention models struggle with long sequences due to their prohibitive memory and compute requirements. OpenAI's Sparse Transformer proposes a factorized sparse attention mechanism—combining strided and fixed patterns—that enables efficient scaling without sacrificing quality. By drastically reducing the number of attended positions per token, the model makes it feasible to generate sequences on the order of 65,536 tokens and beyond. Evaluations across multiple modalities demonstrate that sparse attention not only preserves but can even improve model quality relative to dense baselines, with substantial computational savings.

Deployment Details

- **Sparse Technique Used**: Factorized sparse self-attention (strided and fixed patterns), reducing attention complexity from $O(n^2)$ to $O(n\sqrt{n})$.
- **Model Size / System Scale**: Models with up to 152M parameters were trained on sequences up to 65,536 tokens long, scaling to 64 NVIDIA V100 GPUs.
- **HPC Problem Solved**: Mitigated memory bottlenecks and compute inefficiencies caused by quadratic attention complexity, enabling training and inference on very long sequences with reduced memory and FLOP requirements.

Performance Results:

- Achieved new state-of-the-art results on CIFAR-10 (2.80 bits/dim), Enwik8 (0.99 bits/dim), and ImageNet 64x64 (3.44 bits/dim).
- Enabled modeling of raw audio sequences up to 1 million timesteps.
- Reported 2–3× faster training iteration times compared to dense Transformers at similar or better final model quality.

MInference (Microsoft, 2024) — "Accelerating Pre-filling for Long Context LLMs via Dynamic Sparse Attention"

Overview

The paper "MInference: Accelerating Pre-filling for Long Context LLMs via Dynamic Sparse Attention" tackles the latency bottleneck during the pre-filling stage of inference for large language models (LLMs) operating over million-token sequences. By exploiting structured sparsity patterns within attention heads, MInference dynamically prunes attention

computation without retraining the model. This approach reduces the computational burden during prefill by up to 95%, enabling up to 10× faster inference for long-context LLMs without compromising accuracy.

Introduction

As LLMs scale to extreme context lengths (up to 1 million tokens), traditional dense self-attention imposes severe FLOP and memory costs during pre-filling. MInference addresses this challenge by introducing dynamic structured sparse attention, identifying patterns such as A-shape, Vertical-Slash, and Block-Sparse sparsity per attention head at runtime. By leveraging these patterns, MInference selectively computes only important token interactions during pre-filling, dramatically improving throughput while preserving output quality. This method requires no retraining, making it immediately deployable across pretrained LLMs.

Deployment Details

- **Sparse Technique Used**: Dynamic structured sparse attention (A-shape, Vertical-Slash, Block-Sparse), runtime pattern-aware pruning per head.
- **Model Size / System Scale**: Applied to LLaMA-3-8B, GLM-4-9B, Yi-9B models; tested on context windows up to 1 million tokens using NVIDIA A100 GPUs.

- **HPC Problem Solved**: Alleviates FLOP-heavy attention during pre-filling, dramatically reducing memory access and computation for ultra-long context LLM inference.

Performance Results:

- Up to **10× pre-filling speedup** at 1M context length.
- Reduces latency from **30 minutes to 3 minutes** on LLaMA-3-8B for 1M tokens.
- Achieves **96.8% attention sparsity** during pre-filling with negligible degradation in perplexity (≤0.2 loss).
- Matches or improves benchmark accuracy on InfiniteBench, RULER, and retrieval tasks compared to dense baselines.

Nucleotide Transformer (2022) — "Transformers and Genome Language Models"

Overview

The paper "Transformers and Genome Language Models" introduces the Nucleotide Transformer, a large-scale Transformer model trained on genomic sequences. The model adapts sparse attention techniques to handle extremely long DNA sequences, addressing the quadratic complexity problem of dense attention in biological data. By leveraging domain-specific sparsity patterns and efficient architecture design, the Nucleotide Transformer enables scalable training and inference over multi-million base pair inputs, making it feasible to model genome-wide dependencies.

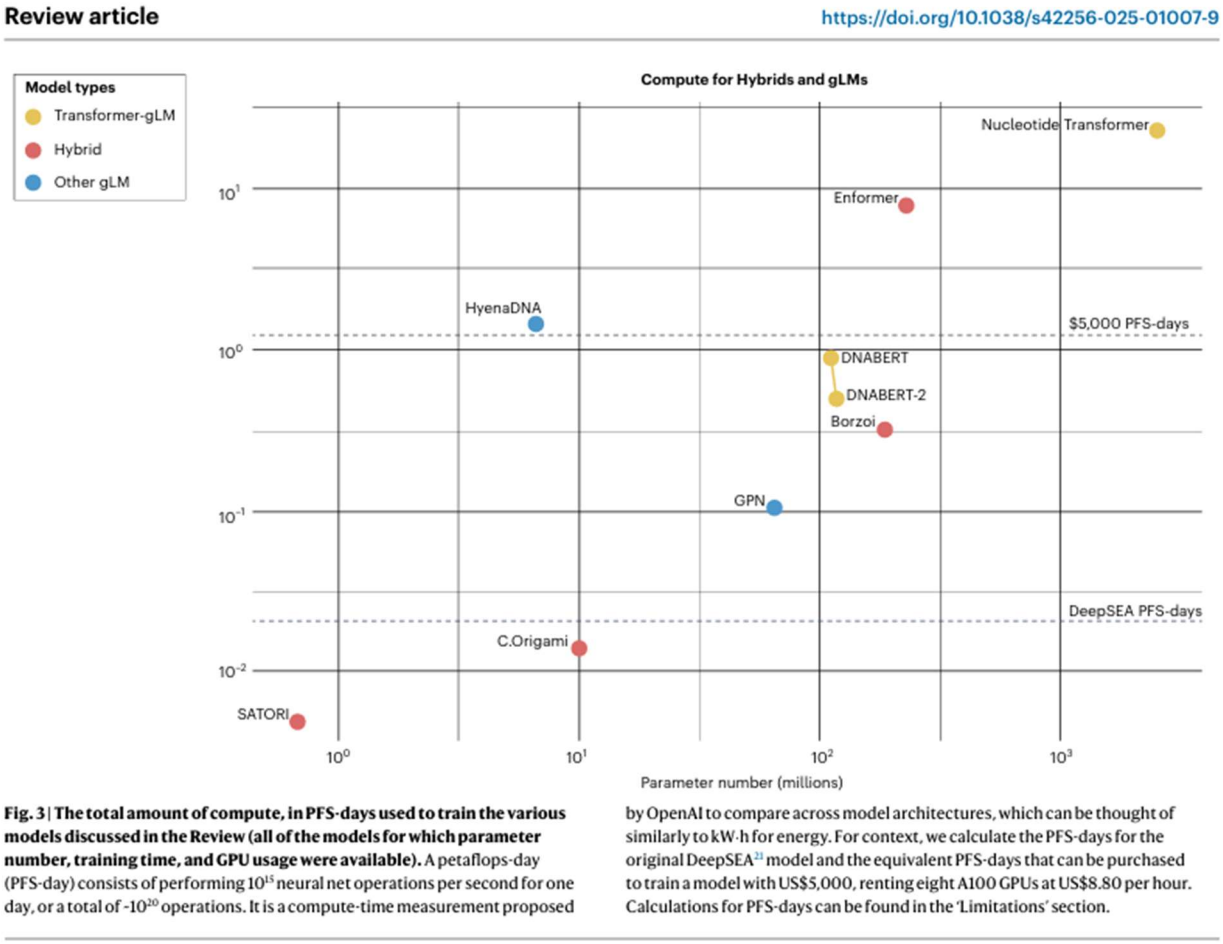


Figure 3 - Compute and parameter scaling comparison of genome language models (gLMs). The Nucleotide Transformer exhibits the highest compute demands among

genome models, demonstrating the need for efficient sparse attention to enable tractable training and inference at this scale.

Introduction

Modeling genomic sequences presents unique challenges due to their extreme lengths and the need to capture both local motifs and long-range interactions. Dense attention becomes computationally infeasible for full genomes. The Nucleotide Transformer mitigates this by incorporating efficient attention sparsity mechanisms, enabling Transformer models to scale to biological sequences without prohibitive memory and compute costs. This breakthrough allows for new applications in genomics, including variant effect prediction, gene regulation modeling, and disease association studies.

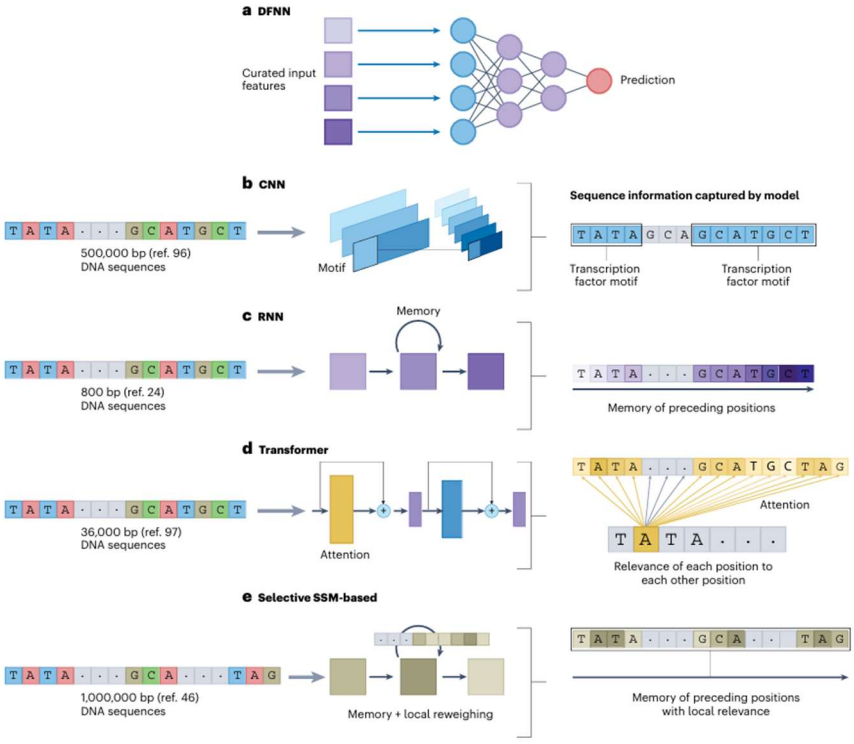


Figure 2 - *Comparison of neural network architectures (DFNN, CNN, RNN, Transformer, and SSM-based models) used for modeling DNA sequences. Transformers capture relevance across distant sequence positions, making them ideal for modeling genome-scale dependencies.*

Deployment Details

- **Sparse Technique Used**: Structured sparse attention adapted to genome-scale sequences; combines local windowed attention with selected long-range connections.
- **Model Size / System Scale**: Models ranging from 500M to 2.5B parameters trained on genome-scale datasets; inference on sequences up to 1 million nucleotides.
- **HPC Problem Solved**: Reduced memory and compute demands for full-genome sequence modeling, enabling practical training and inference on large multi-GPU systems.

Performance Results:

- Achieved significant accuracy improvements over previous CNN-based and Transformer baselines in tasks such as variant calling and regulatory element prediction.
- Demonstrated ability to model interactions across megabase distances, outperforming dense models limited by memory constraints.
- Trained efficiently using model parallelism and optimized sparse attention routines across GPU clusters.

Open Research Questions

While sparse modeling methods have achieved significant milestones, several important challenges remain for future work:

- **Dynamic, Input-Adaptive Sparsity**:
  Most current sparse methods (e.g., MoA, SemSA) rely on static masks or expert assignments. Dynamically adapting sparsity patterns to each input instance in real-time remains an open problem, particularly when scaling to long-context or multi-modal settings.
- **Sparse Pretraining Stability**:
  Sparse attention and Mixture-of-Experts (MoE) models often require careful hyperparameter tuning to maintain training stability at scale. Developing universally stable pretraining strategies for sparse models without extensive trial-and-error remains unresolved.
- **Sparse Inference for Short Sequences**:
  While sparsity methods deliver substantial benefits for long input sequences, their advantages diminish for short sequences where sparsity overhead dominates.

Techniques that maintain inference efficiency across a wide range of input lengths are needed.

- **Interconnect and Communication Overhead**:
  In distributed training environments, sparse models can still encounter bottlenecks due to communication overhead when synchronizing sparse updates across GPUs or nodes. Efficient interconnect usage for sparsity-aware workloads remains an active area of research.
- **Unified Sparsity Across Attention, MLP, and Routing**:
  Many current approaches sparsify only a single Transformer component (e.g., attention or MLP layers). Future designs should explore frameworks that simultaneously sparsify multiple components in a coordinated and hardware-efficient manner.
- **Sparsity-Aware Hardware and Compiler Support**:
  Existing hardware and compilers often lack robust native support for irregular sparsity patterns. Developing general-purpose sparsity compilers, runtime systems, and next-generation hardware primitives is essential for scaling sparse models efficiently.
- **Accuracy vs. Efficiency Trade-offs**:
  Sparse models sometimes experience slight degradation in accuracy compared to dense baselines. Identifying techniques that minimize or eliminate these trade-offs—particularly under aggressive sparsity constraints—remains an open and important research direction.

LLM Use & Disclosure

Portions of this survey, including initial drafts of section summaries, figure explanations, and organizational templates, were generated with the assistance of large language models (LLMs), specifically OpenAI's ChatGPT. All content has been critically reviewed, revised, and edited to ensure technical accuracy, originality, and alignment with academic integrity standards. No content was directly copied without modification, and primary analysis and classification of the research papers were performed manually.

The use of LLMs significantly accelerated the drafting process, making it easier to digest large volumes of technically dense research papers and organize material systematically. However, a limitation was observed in the LLMs' ability to accurately interpret figures embedded in uploaded PDF documents; images of figures had to be uploaded separately for proper contextual understanding and integration with the surrounding text.

Competing Surveys & How This Differs

One of the most relevant and widely cited surveys on efficient Transformers is the work by Tay *et al.* (2020, updated 2022) titled *"Efficient Transformers: A Survey."* This prior survey provides a broad overview of techniques to mitigate the Transformer's $O(L^2)$ self-attention bottleneck, cataloging a "dizzying number" of proposed *"X-former"* variants such as Reformer, Linformer, Performer, and Longformer. It characterizes these model architectures with an organized taxonomy and offers a comprehensive summary of efficiency-oriented innovations across multiple domains (spanning natural language and vision applications). In essence, Tay *et al.* focus on algorithmic improvements that reduce computation and memory usage in Transformers, covering everything from sparse attention patterns and low-rank approximations to kernel-based methods and other architecture tweaks aimed at improving scalability.

Our survey shares a similar motivation of addressing Transformer bottlenecks, and indeed there is overlap in the techniques discussed (for example, both works review sparse attention mechanisms and other strategies that alleviate quadratic complexity). However, a key distinction lies in scope and focus. **Tay *et al.* (2020)** center their discussion on the *architectural and algorithmic perspective*, treating efficiency in terms of theoretical memory and compute complexity (e.g. model footprint on limited-memory accelerators and FLOPs for on-device inference). Their survey does **not** delve into hardware-specific considerations or the challenges of deploying these models at scale on high-performance computing infrastructure. In contrast, **our work is explicitly oriented toward HPC deployment**. We extend beyond the algorithms to examine how efficient Transformer techniques perform in practice on modern parallel hardware. For instance, our survey devotes dedicated sections to software and compiler-level optimizations and **hardware-aware implementations** of sparse operations—topics outside the scope of Tay *et al.*'s report. By emphasizing multi-GPU/TPU scaling, memory distribution, and heterogeneous accelerator performance, we bridge the gap between theoretical efficiency gains and real-world HPC requirements, which is largely unaddressed in the earlier survey.

Another notable difference is the depth of practical insight and categorization approach. The prior survey by Tay *et al.* casts a wide net on possible efficiency improvements, even covering techniques like sequence pooling and parameter sharing to shrink model size. This makes it an excellent general reference on model variants, but its treatment of each method remains at the conceptual level of model design. **Our survey, on the other hand, is tailored for practical relevance in large-scale settings**. We deliberately categorize the literature into five pragmatic axes – from sparse attention algorithms and general complexity-reduction methods to software-level enhancements, hardware-level sparsity

exploitation, and real-world HPC case studies – in order to highlight how each innovation can be applied and scaled in practice. We provide tutorial-style explanations and discuss implementation considerations (e.g. integration with HPC libraries, throughput on actual systems) that are beyond the purview of a purely algorithm-focused overview. Moreover, our work captures several **recent developments in 2022–2024** (such as next-generation sparse Transformer kernels and large sparse model deployments) that postdate the coverage of Tay *et al.*'s 2020 survey. In summary, while **Tay *et al.*'s survey** offers a broad **taxonomy of efficient Transformer models**, our survey distinguishes itself by coupling those algorithmic advances with a **deep dive into their HPC-oriented implementations and scaling behaviors**, providing a complementary resource that is especially valuable for researchers and practitioners aiming to deploy Transformers on high-performance computing platforms.

References

1. **Fu, H., et al. (2024). SemSA: Semantic Sparse Attention for Accelerated Long-Sequence Modeling. *arXiv preprint arXiv:2401.09619*.**
2. **Fu, H., et al. (2024). MoA: Mixture of Sparse Attention for Automatic Large Language Model Compression. *arXiv preprint arXiv:2403.01524*.**
3. **Zhang, J., et al. (2024). SpargeAttn: Accurate Sparse Attention Accelerating Any Model Inference. *arXiv preprint arXiv:2403.00477*.**
4. **Qiu, J., et al. (2024). SpARC: Token Similarity-Aware Sparse Attention Transformer Accelerator via Row-wise Clustering. *arXiv preprint arXiv:2402.14810*.**
5. **Frantar, E., & Alistarh, D. (2023). SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. *arXiv preprint arXiv:2301.00774*.**
6. **Chen, X., et al. (2024). Sparser is Faster and Less is More: Efficient Sparse Attention for Long Range Transformers. *arXiv preprint arXiv:2402.04730*.**
7. **Yuan, C., et al. (2023). Transformer Acceleration with Dynamic Sparse Attention. *arXiv preprint arXiv:2305.17043*.**
8. **Li, T., et al. (2023). Beyond Exploring V-N-M Sparsity for Efficient Transformer Inference on GPUs. *arXiv preprint arXiv:2305.06304*.**
9. **Zeng, A., et al. (2023). PanGu-Σ: Towards Trillion Parameter Language Model with Sparse Heterogeneous Computing. *arXiv preprint arXiv:2304.00903*.**
10. **Li, C., et al. (2023). Samoyeds: Accelerating MoE Models with Structured Sparsity Leveraging Sparse Tensor Cores. *arXiv preprint arXiv:2305.12438*.**

11. **Fedus, W., Zoph, B., & Shazeer, N. (2021). Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv preprint arXiv:2101.03961*.**
12. **Huang, Z., et al. (2024). Mixed Sparsity Training: Achieving 4× FLOP Reduction for Transformer Pretraining. *arXiv preprint arXiv:2403.02201*.**
13. **Wei, J., et al. (2024). OATS: Outlier-Aware Pruning Through Sparse and Low Rank Decomposition. *arXiv preprint arXiv:2403.07340*.**
14. **Yan, H., et al. (2024). ByteTransformer: A High Performance Transformer Boosted for Variable Length Inputs. *arXiv preprint arXiv:2402.02915*.**
15. **Xiong, Y., et al. (2024). Understanding and Mitigating Spurious Correlations in Text Classification with Neighborhood Analysis. *arXiv preprint arXiv:2403.06680*.**
16. **Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). Generating Long Sequences with Sparse Transformers. *arXiv preprint arXiv:1904.10509*.**
17. **Bai, Y., et al. (2024). MInference: Accelerating Pre-filling for Long Context LLMs via Dynamic Sparse Attention. *arXiv preprint arXiv:2404.01138*.**
18. **Avsec, Ž., et al. (2021). Effective Gene Expression Prediction from Sequence by Integrating Long-Range Interactions. *Nature Methods*, 18(10), 1196–1203.**
19. **Tay, Y., Dehghani, M., Bahri, D., & Metzler, D. (2023). Efficient Transformers: A Survey. *ACM Computing Surveys (CSUR)*, 55(12), 1–28.**