LLM-Guided Autoscheduling for Large-Scale Sparse Machine Learning

Rubens Lacouture
Stanford University
rubensl@stanford.edu

Genghan Zhang
Stanford University
zgh23@stanford.edu

Konstantin Hossfeld
Stanford University
hossfeld@stanford.edu

Tian Zhao
Classie AI
tian@classie.ai

Kunle Olukotun
Stanford University
kunle@stanford.edu

Abstract

Optimizing sparse machine learning (ML) workloads requires navigating a vast schedule space. Two of the most critical aspects of that design space include *which operators to fuse* and *which loop/dataflow order* to use within each fused region. We present AUTOSPARSE, an LLM-guided autoscheduler atop a fusion-capable sparse ML compiler that focuses on *fusion grouping* and *legal dataflow order* selection. The compiler enumerates lawful orders per fused region and exposes a lightweight FLOPs/byte signal; the LLM proposes structured candidates (fusion sets and orders) that we validate and rank before codegen. With backend defaults for blocking and parallelism held fixed, case studies on GCN, GraphSAGE show consistent gains over unfused baselines and parity with hand-tuned/heuristic schedules. Coupling LLM reasoning with compiler legality and roofline-style signals efficiently explores sparse scheduling spaces with minimal human effort.

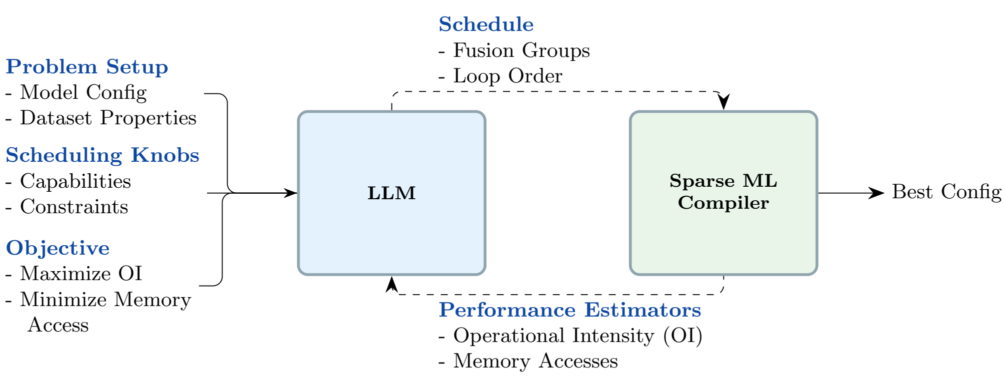

Figure 1: Overview of AUTOSPARSE: the LLM proposes schedules; the compiler validates, enumerates legal dataflow orders, and returns cost estimates; the best configuration is then selected.

1 Introduction

Sparse deep learning models are hard to optimize because of irregular memory access and vast scheduling spaces. High performance usually requires expert decisions about which operations to

Submitted to 39th Conference on Neural Information Processing Systems (NeurIPS 2025). Do not distribute.

fuse to cut memory traffic, which dataflow order to use, how to tile to fit fast memory, and how to parallelize. Manual exploration is infeasible for full models. Autoschedulers in dense settings (e.g., Halide, TVM) have been successful [12, 2], but sparse models add combinatorial dataflow choices and sparsity-specific constraints that make search harder [11, 1, 13]. Dataflow accelerators for sparse workloads further increase the payoff of getting fusion and dataflow order right [8, 7].

Fusion—especially cross-expression fusion in sparse workloads—is a first-order lever for performance. By co-iterating producers/consumers and avoiding materialized intermediates, the right fusion granularity increases operational intensity and can change the algorithmic cost (work and I/O), yielding asymptotic efficiency gains when traversal aligns with sparse storage format [15].

We propose to leverage LLMs to co-pilot the scheduling process. Our *LLM-guided autoscheduler* works with a *fusion-capable sparse ML compiler* targeting dataflow hardware. The compiler can fuse any subset of expressions given a user *Fuse* schedule, then enumerates all *legal* dataflow orders for the fused regions and exposes a lightweight heuristic that estimates FLOPs and memory traffic of alternative plans. The autoscheduler feeds a textual description of the compute graph, the compiler's knobs (fuse sets and dataflow orders), and hardware hints to an LLM, asking for concrete schedule proposals and reasoning. Candidate schedules are validated against compiler invariants and scored using the heuristic; poor candidates are pruned before code generation.

LLMs have shown promise in compiler optimization by reasoning over large discrete spaces and leveraging prior knowledge [9, 6, 3]. For sparse ML, schedule quality hinges on long-range trade-offs (e.g., partial fusion to avoid recomputation versus full fusion to reduce memory traffic), which LLMs can articulate, while the compiler ensures legality and supplies fast cost signals. We evaluate on GCN [10] and GraphSAGE [5], showing robust gains and workload-adaptive fusion choices.

Our contributions are:

- An LLM-guided autoscheduler for sparse ML that wraps a fusion-capable sparse ML compiler able to perform arbitrary expression fusion and human-in-the-loop dataflow selection.
- A semi-structured schedule format that the LLM emits and the compiler validates, plus a cost-guided pruning loop using a FLOPs/bytes heuristic.
- An empirical study on sparse models on a simulated dataflow architecture, demonstrating consistent speedups over unfused baselines and parity with hand-tuned/heuristic strategies.

2 Background and Related Work

Dense autoscheduling has matured in systems such as Halide and TVM [12, 2]. Sparse compilers expose formats and scheduling but mostly target CPUs/GPUs and single-expression fusion [11, 1, 13]. Dataflow abstractions and recent sparse-to-dataflow compilers highlight the importance of fusion and dataflow ordering on streaming hardware [8, 7]. Sequence models motivate block-sparse patterns and specialized attention mechanisms [14, 4]. Our work complements this landscape by using an LLM to search the fuse/order space while keeping the compiler itself a black box that guarantees correctness and provides cost signals.

3 Method

3.1 Problem Setting and Interface

We assume a fusion-capable sparse ML compiler that: (i) ingests a model graph (e.g., PyTorch) and a user *Fuse* schedule; (ii) runs its internal fusion then *enumerates legal dataflow orders* for each fused region; (iii) provides a fast FLOPs/bytes heuristic for any (fusion, order) pair; and (iv) generates code for a dataflow backend once a schedule is selected.

3.2 LLM-Guided Autoscheduling Loop

Prompt state. We serialize operator types, tensor shapes, sparsity statistics, producer–consumer relations, and known bottlenecks (via coarse roofline classification). In addition, the prompt state captures admissible fuse sets and compiler-enumerated legal execution orders for each fused region. (Tiling and parallelization are treated as *context only* and are not selected by the LLM.)

Proposals and validation. At each iteration, the LLM (i) identifies bottlenecks, (ii) picks a fusion granularity, and (iii) selects a *legal* dataflow order *from* the compiler's set. Outputs use a compact schema; we validate names/shapes and that every order is compiler-legal, then score with the compiler's FLOPs/bytes heuristic. Candidates are ranked by a roofline-style score and pruned; if none pass thresholds, we request alternatives.

Proposal schema. The LLM communicates via a compact JSON-like schema (Listing 1).

Listing 1: AUTOSPARSE proposal schema.

```
Plan := {"rank": int, "score": number, "estimated_OI": number,
        "fusion_groups": [ {"name": string,
                           "ops": [string],
                           "dataflow_order": [string]} ]}
```

3.3 Heuristic and Search Pruning

The heuristic symbolically estimates FLOPs and bytes for a fused loop nest, computes operational intensity (FLOPs/byte), and classifies compute- vs. memory-bound relative to machine balance. It rewards elementwise fusion and penalizes plans that increase FLOPs via recomputation. Coupled with legal-order enumeration, this yields a small, high-quality candidate set.

3.4 Human-in-the-Loop Selection

When multiple near-ties remain, we surface top-k plans with LLM rationales and heuristic estimates; practitioners may override choices—typically dataflow order—based on dataset-specific sparsity.

3.5 Why LLM-guided vs. solver- or BO-based search

Sparse scheduling spans fusion grouping and legal dataflow orders, forming a large, discrete, hierarchical space with non-smooth objectives (memory-fit thresholds, sparsity-dependent reuse). Exact solvers (ILP/CP/SMT) need brittle encodings and scale poorly; Bayesian Optimization assumes smooth, low-dimensional objectives and struggles here. We instead pair the compiler—which enumerates legal orders and provides a fast FLOPs/bytes signal—with an LLM that proposes structured fusion+order plans; a validator enforces legality and a roofline-style score prunes candidates, yielding high sample efficiency. Once structure is fixed, BO/small solvers can tune numeric knobs offline (e.g., tile/parallel factors).

4 Evaluation

Workloads. Two-layer GCN and GraphSAGE on real-world datasets (Table 1).
Backend. Configurable dataflow simulator calibrated to an FPGA (Xilinx Virtex UltraScale+ VU9P); backend blocking and parallelization are held fixed to isolate fusion/dataflow order effects.
Implementation. AUTOSPARSE queries the compiler's capabilities and constraints, proposes schedules via the schema in Listing 1, and runs a short beam search (typical: 5–15 iterations, beam 6–16).
Baselines. (i) UNFUSED: compiler default with no cross-op fusion; (ii) FULLY FUSED: compiler's greedy fusion/order; (iii) HAND: expert/curated schedule.
LLM configuration. We used GPT-5 Thinking (OpenAI) to generate schedule proposals. The model's outputs followed Listing 1's schema and were validated for legality before scoring.

The evaluation on two-layer GCN and GraphSAGE workloads shows that AUTOSPARSE achieves essentially the same performance as the hand-crafted expert schedule. In particular, the reported geomean speedup over a no-fusion baseline is about 1.85x for GCN and 2.22x for GraphSAGE (about 2x overall). By contrast, the fully-fused (greedy) schedule performs very poorly – the LLM-guided plan runs roughly 13-15x faster (geomean) than the fully-fused case – indicating that indiscriminate full fusion greatly inflates memory traffic and hurts performance.

As shown in Figure 3, the fused schedules found by AUTOSPARSE preserve the same total FLOPs as unfused execution but dramatically cut memory traffic. For example, on the largest dataset (OGB-Collab) the GCN model has 1.7 GFLOPs in both cases but bytes drop from 445.9 MiB (unfused) to

186.8MiB (best fused); similarly, GraphSAGE on OGB-Collab goes from 625.2 MiB to 316.4 MiB. This roughly 2x reduction in data movement (with FLOPs unchanged) underlies the speedup gains. The LLM's search effort is modest (Table 2): per workload, the beam search ran only 9–11 iterations and tested about 28–36 candidate schedules. In other words, only a few dozen configurations were evaluated before converging on the expert-equivalent schedule, suggesting the LLM proposals were close to optimal.

Table 1: Structure of the graph datasets.

Category	Metric	Cora	Cora_ML	DBLP	OGB-Collab
	#Vertex	2708	2995	17716	235868
Structure	#Edge (directed)	10556	16316	105734	2570930
	Feature length	1433-16-7	2879-16-7	1639-16-4	128-16-2

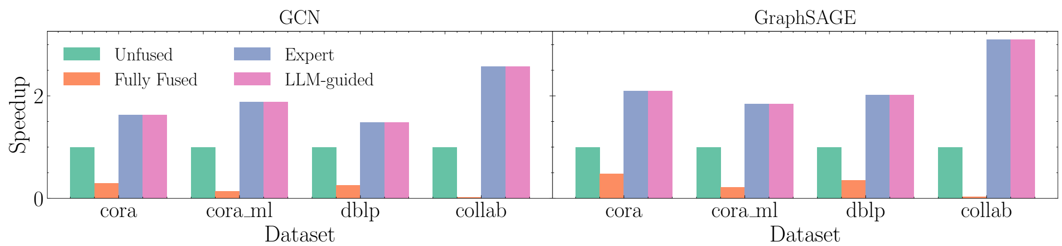

Figure 2: End-to-end speedup (vs. Unfused) for GCN and GraphSAGE on various datasets. The LLM-guided schedule matches the human-expert selection across all datasets and outperforms the unfused and fully fused baselines. Geomean speedup vs. Unfused: **1.85**× (GCN) and **2.22**× (GraphSAGE).

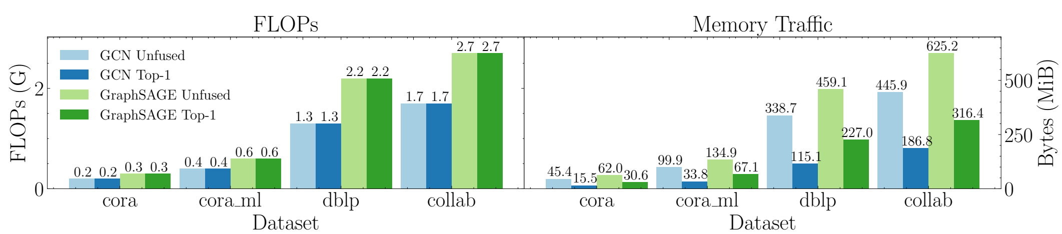

Figure 3: FLOPs (G) and Bytes (MiB) per forward pass. Unfused = no fusion; Top-1 = best fused configuration from the LLM search.

Table 2: Autoscheduler search statistics for GCN and GraphSAGE.

Model	Metric	Cora	Cora_ML	DBLP	OGB-Collab
GCN	Iterations	9	10	11	10
	Tested points	28	30	34	36
GraphSAGE	Iterations	9	10	11	11
	Tested points	28	30	32	36

5 Conclusion

AUTOSPARSE couples an LLM's structural proposals with compiler legality and a cheap cost signal to traverse sparse scheduling spaces. Results across representative workloads indicate the approach is practical and competitive with expert schedules under fixed backend settings.

References

[1] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in mlir. *ACM Trans. Archit. Code Optim.*, 19(4), September 2022.

[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[3] Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Llm compiler: Foundation language models for compiler optimization. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction*, pages 141–153, 2025.

[4] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc.

[6] Charles Hong, Sahil Bhatia, Altan Haan, Shengjun Kris Dong, Dima Nikiforov, Alvin Cheung, and Yakun Sophia Shao. Llm-aided compilation for tensor accelerators. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–14. IEEE, 2024.

[7] Olivia Hsu, Alexander Rucker, Tian Zhao, Varun Desai, Kunle Olukotun, and Fredrik Kjolstad. Stardust: Compiling sparse tensor algebra to a reconfigurable dataflow architecture. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, CGO '25, page 628–643, New York, NY, USA, 2025. Association for Computing Machinery.

[8] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S Emer, Mark A Horowitz, and Fredrik Kjølstad. The sparse abstract machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 710–726, 2023.

[9] Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W Mahoney, Kurt Keutzer, and Amir Gholami. An llm compiler for parallel function calling. In *Forty-first International Conference on Machine Learning*, 2024.

[10] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017.

[11] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–29, 2017.

[12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4), July 2012.

[13] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. Sparsetir: Composable abstractions for sparse compilation in deep learning, 2022.

[14] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

[15] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. Compilation of modular and general sparse workspaces. *Proceedings of the ACM on Programming Languages*, 8(PLDI):1213–1238, 2024.